Software portability

Thesis

UNRESTRICTED

ISSUE 3

SUBMISSION TO THE OPEN UNIVERSITY AS A THESIS FOR THE DEGREE OF MASTER OF PHILOSOPHY

SOFTWARE PORTABILITY

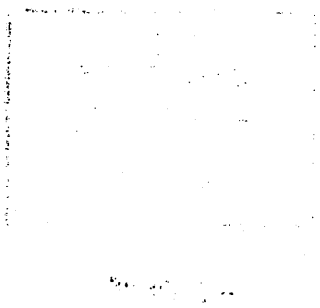
AUTHOR : R P SMITH

Date of Submission: February 1985
Date of Award: 4.8.86

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

CONTENTS

ACKNOWLEDGEMENT
I gratefuly acknowledge the help of Mr M Taylor in the design of the Case operations of the Ivor abstract machine. I acknowledge also the assistance of my OU tutor, Dr Max Bramer.



1. INTRODUCTION

This thesis represents the submission to the Open University for the degree of Master of Philosophy.

The research topic lies within the discipline of computer science and in particular problems associated with the transportability of computer software are investigated.

The study period was organised into 3 parts :

- A literature search to look at what work had been done in the area.

- The design of an intermediate language for a compiler.

- The specification, design and implementation of a functional programming language in order to develop a portable programming environment.

This thesis covers the work of the period Sep78 to Jun82, all of which was performed by part time study.

The Thesis comprises 4 major sections and an appendix which shows program listings. The sections are as follows :

- Section 1 : The introduction

- Section 2 : The result of a literature search.

- Section 3 : A design for a compiler intermediate language, Ivor, covering the architecture of the underlying abstract machine and the structure of the intermediate language.

- Section 4 : The design and implementation of a function based programming language, Mule, including the structure of the language and a description of the implementation.

2 CONCEPTS OF PORTABILITY

2.1 The Meaning of Portability

In electronics hardware it is not surprising to talk about 'plug compatible' components developed by different manufacturers, performing identical functions and being completely interchangeable. Software has been treated differently and it is generally accepted that a program running on one computer will not run automatically on another.

The problem of having 'plug compatible' software is usually one of specification. Hardware components can be specified (almost) precisely and the language used to specify the component and its environment are based upon basic scientific or engineering laws and with well understood parameters. For example : operating temperature, voltage levels, capacitance, ohmic resistance, frequency of applied signals, etc.

A software item is usually specified in terms of : the functions the program is to perform, the acceptable inputs, the acceptable outputs, performance requirements, reliability requirements and (sometimes) environmental constraints. Acceptable inputs are often defined in some form of BNF with semantics defined informally as English prose. Rarely are all of the possible outputs defined, particularly error cases which are left to the software designer to invent. The remainder of the specification is usually written in an informal way and some of the quantitative restrictions are not documented at all.

The problem of transportability of software lies with the specification which inadequately addresses the requirements of the program and the definition of the operating environment.

The following sections will look at some of the documented methods of reducing the problems associated with porting programs, in particular : high level programming languages, universal assemblers and abstract machines, specification methods.

2.2 Transporting Programs

A customer usually does not want a program transported from one environment to another, though he may express his requirements in those terms. What the customer wants is an (almost) identical set of facilities on the new environment. In [Reference 3 : Brown (Editor)] a program is defined as portable if the effort required to move the program is significantly less than that required to rewrite it. It seems that in practice if more than 20% of a program requires rewriting it is preferable to start again, unless the software is very well structured.

Programs are not automatically transportable and portability needs to be designed into the program. The following sections

look at methods which may be employed to make programs more portable.

2.2.1 High Level Languages

The high level language is an important tool used in the production of software. The disciplined use of a suitable high level language ensures the production of a structured and well produced program which can be supported and enhanced. In addition a program written in a high level language can be (theoretically) transported to any computer system for which there are language support facilities.

There are many high level languages ranging from simple languages like Fortran, Basic and Cobol through to systems type languages like BCPL, BLISS, Lisp and the block structured languages like Coral 66, Algol-68, Pascal, Ada and Chill. Some of these languages are specified rigorously but many are not. Techniques for specifiying languages better and with more precision are becoming available. This is discussed further in [Reference 2 : Lucas & Lauer 1970]. In addition mathematically based 'functional languages' aimed at specifying systems fully are appearing [Reference 1 : Backus 1978] and [Reference 37 : Allen]. The lack of rigour tends to produce differing interpretations by language support implementors for some of the features of the language. As a consequence a facility which operates in one way on one machine may operate in a different way on another.

As an example of this figure 2.2.1-1 shows a part of a CORAL-66 program which has,in fact, been designed to be portable. Even so, it is unlikely that the program will operate without changes on another computer for the following reasons :

- The procedure uses names which are greater than 12 characters in length. The definition of CORAL-66 states that names are significant to the first 12 characters but some implementations restrict the implementation further allowing less characters for external-names and for macro-expansion names.

- On line 109 (and elsewhere) the statement 'IF' ALPHABETIC M(CHAR IL) ... disguises in the macro call that the internal character set of the underlying machine is being used to determine a condition, in this case the ICL-George 3 6bit code. The macro definition would require changing if the program were moved to, say, an ASCII or EBCIDIC character set machine.

- On line 162 the character string uses "visible spaces" which are expected by this particular compiler. The compiler removes all spaces, including those within text strings. As a consequence of this the "%" character needs to be used within a text string in place of a space. This is not the

case with all compilers.

- On lines 167 to 169 the procedure sets up a character string which uses an integer to store the character string's location. The character string packing and internal character set, including the storage of the length of the character string, assumes a particular compiler storage allocation strategy.

- On line 257 the code assumes a 6-bit byte, as do lines 317, 318 and 319 though the use of a 6-bit byte is disguised by the use of macros.

Figure 2.2.1-1

```
002  PROCEDURE fetch element pg;
098  BEGIN
099  INTEGER ARRAY element dl[0:char lit length m];
100  INTEGER char il,kl,hash il,string delimiter il;

103  COMMENT read the element;

105  char il := read a rel char pg;
106  element dl[1] := char il;
107  kl := 2;

109       IF alphabetic m(char il) OR numeric m(char il) THEN
112       BEGIN
114         COMMENT the case of an element being an alpha-numeric
               name or a literal value;
118            FOR char il := read a rel char pg
               WHILE alphabetic m(char il) or numeric m(char il) DO
121            BEGIN
123                 IF irrelevant skipped bg=true m THEN GOTO out1 ll;
125              element dl[kl] := char il;
126              inc m(kl,1);
127            END;
129     out1 ll:
130       END ELSE
132       BEGIN
134         COMMENT case of the element being an operator type (ie
               non alpha-numeric) name or character string;
139         COMMENT check first for character string;
141         string delimiter il := char il;
143            IF string delimiter il = char lit 1 delimiter m OR
144               string delimiter il = char lit 2 delimiter m THEN
146            BEGIN
148              COMMENT this is a character string literal;
151              COMMENT fetch the string and return it ;
153              kl := 1;
155               FOR  char il := ich m
                  WHILE char il <> string delimiter il DO
157               BEGIN
158                 element dl[kl] := char il;
159                 inc m(kl,1);

161                 IF kl>char lit length m THEN
162                 errorhandler(LITERAL(e),
                    "Character%string%literal%is%too%long");

164                 IF char il=new line m THEN
                    inc m(current line ig,1);
166             END;
.he                  Figure 2.2.1-1 (Continued)
167             element dl[0]:=kl-1; (Character string length)
168             element type ig := ele char lit m;
169             element ig := LOCATION(element dl[0]);
```

```
170             return m;
171           END;

173         FOR  char il := read a rel char pg
            WHILE non alpha numeric m(char il) DO
175         BEGIN
177           IF irrelavant skipped bg=true m THEN GOTO out211;
179           element dl[kl]:=char il;
180           inc m(kl,1);
181         END;

183       out211:
184       END;

185 back step m;
186 char buff mark ig := char buff mark ig-1;
187 element dl[0] := kl-1;

188 COMMENT at this stage the array 'element dl' is set up  such
    that  element-0  is the character length of the element  and
    element-1  to  element-n  contain  the  characters  of  the
    element;

195 IF numeric m(element dl[1]) THEN
197 BEGIN
199   COMMENT case of a numeric constant being returned ;

201   kl := element dl[0]+1;
202   element ig := 0;

204   FOR kl:=kl-1 WHILE kl>0 DO
206   BEGIN
207     element ig := element ig+power of 10 dg[
        element dl[0]-kl]*element dl[kl];
208   END;

209   element type ig := ele literal m;
210   return m;

211 END ELSE
213 BEGIN
215 COMMENT  this  is the complex part of the procedure where  a
    'name' is to be returned.  Firstly it is necessary to search
    the  object  table to see if the name  has  been  introduced
    already;
221   hash il := 0;
225 search for name ll:

227   IF object string dg[hash il]<> empty m THEN
229   BEGIN
231     COMMENT compare string table with the elements stored
        in 'element dl';
235     FOR kl:=0,kl+1 WHILE kl<= object string dg[hash il] DO
237     BEGIN
```

```
239         IF element dl[kl]<> object string dg[kl+hash il] THEN
241         BEGIN
243           COMMENT the name is not the element so try the next
                name;
245           hash il := hash il+4+object string dg[hash il];
246           repeat m(search for name ll);
247         END;
248       END;

250     COMMENT this is the element name so fetch the object
        table index and return it in 'element ig' whilst
        setting 'element type ig' to 'ele operand m';
256     hash il:=hash il+object string dg[hash il]+1;
257     element ig:=object string dg[hash il]*64*64 +
                    object string dg[hash il+1]*64 +
                    object string dg[hash il+2];
259     element type ig := el operand m;

261     IF in intro bg <> false m AND
           element ig <> rat dot m AND
           element ig <> rat colon m THEN
262     errorhandler pg(LITERAL(e),
           "Duplicate%introduction%of%an%object");
264     END ELSE
266     BEGIN
268       COMMENT this is the case where the name is not known
          and hence the object must be declared;
273       COMMENT check that the static function nesting
          depth is not >0 as declarations are not
          allowed if this is the case;
277       IF func nest depth ig > 0 THEN
278       errorhandler pg(LITERAL(e),
          "Names%cannot%be%introduced%in%a%nested%function%definition");
```

```
281       COMMENT if this is not an introduction (ie USE ...) warn
          the man that we are introducing an object;

285       IF in intro bg=false m THEN
286       errorhandler pg(LITERAL(w),
          "introduction%of%an%object%assumed");

298       obj size m[max obj table mark ig]:=default obj size m;
                (fill in object size)
300       obj str mark m[max obj table mark ig] := hash il;
                (and the place where the name will be kept as
                 an index into the object string table.  Use
                 the next free object table slot)
306       COMMENT now put the name of the object into the string table;

309       FOR kl:=0,kl+1 WHILE kl<=element dl[0] DO
310       object string dl[hash il+kl]:=element dl[kl];

312       COMMENT and now link back into the object table - into
          3 bytes with the most significant first;
316       inc m(hash il,element dl[0]+1);
317       object string dg[hash il  ]:=byte 2 m(max obj table mark ig);
318       object string dg[hash il+1]:=byte 1 m(max obj table mark ig);
319       object string dg[hash il+2]:=byte 0 m(max obj table mark ig);

321       COMMENT plant code to declare the object;
323       program dg[prog mark ig] := rat declare m;
324       program dg[prog mark ig+1] :=
           byte 2 m(max obj table mark ig);
325       program dg[prog mark ig+2] :=
           byte 1 m(max obj table mark ig);
326       program dg[prog mark ig+3] :=
           byte 0 m(max obj table mark ig);
327       inc m(prog mark ig,4);

329       IF prog mark ig>prog size m THEN errorhandler pg(LITERAL(e),
          "No%program%space%left");

332       COMMENT now set up the return values - firstly 'element ig'
          is set to the object table index of the object and
          'element type ig' is set to 'ele operand m';
337       element ig := max obj table mark ig;
338       element type ig := ele operand m;

340       COMMENT and finally....  tidy up by :
          incrementing 'max obj table mark ig' checking for overflow
          of the object, and object string tables;
344       inc m(max obj table mark ig,2);
346       IF max obj table mark ig>obj table size m OR
347          hash il+2>obj str size m THEN
348       errorhandler pg(LITERAL(e),"Too%many%names%introduced");

349     END;
```

```
350   END;

352   return ll:
353   END fetch element pg;
```

The high level language, however ill defined, can be used effectively in the transfer of software between machine environments provided the language support system used is very similar on each machine. Less effort will be required if there are less differences between the support systems of the different machines. The problem of portability is then tansferred from the software item to the translation or interpretation system. This is discussed further in section 2.3.

The use of a high level language, rather than not using one, greatly improves the chances of being able to transport the software from one computer to another. The large number of high level languages and the lack of rigour in the definition of some languages detracts from the ability to transport programs. The situation is becoming better with the standardisation of many languages, eg COBOL, Fortran 77, UCSD Pascal and latterly with Ada and the introduction of formal methods for defining languages.

In the past only the language syntax was specified rigorously using BNF type constructions. The semantics were defined more loosly either in English, or by example. In addition many languages omitted program segmentation, concurrent process communication and input/output. This leads to a family of languages, each similar but with sufficient differences to make program transportability difficult.

The problems associated with a language definition will be explored with reference to the CORAL 66 language which is a MoD standard language (Reference 4).

- Program segmentation is loosly defined by the COMMON, EXTERNAL and ABSOLUTE communicators. Objects which are introduced in a COMMON communicator may be used in the program segment being compiled with, or without, being declared in that segment.

  EXTERNAL communicators are similar to COMMON communicators except that the items being declared need not be declared in any of the CORAL 66 program segments but may be linked from a subroutine library.

  ABSOLUTE communicators are intended to allow access to absolutely addressed primitive facilities or machine locations of the "underlying machine" of the program being run.

  It is this area of a CORAL 66 program which is particularly loosly defined and implemented in a variety of ways. Most implementations of CORAL 66 require that the introduction of a CORAL 66 place specification, ie a procedure, label or switch, in a COMMON communicator be mirrored by a declaration of the object in the outer block of one program segment. The same is not true for CORAL 66 data

objects, ie integers, fixed decimal, arrays and tables, where the language definition is ambiguous. In most implementations it is required that the data objects be declared in the COMMON communicator only and not in any outer program segment.

The language definition does not define that the COMMON communicator should be identical in each program segment compiled. In some compilers it is essential that this is the case as space for objects is allocated sequentially, relying upon the relative position within a common area for object addresses. In other compilers only COMMON data object introductions need to be identical in each program segment as place specifications are addressed symbolically, ie by name. In other compilers the position of introductions within a COMMON communicator is not important as all addresses are treated symbolically and indeed it is not necessary for communicated objects which are not used in a program segment to be introduced in a COMMON communicator for that segment.

The EXTERNAL communicator is intended to allow the programmer to access operating system routines or procedures which have been written in another language. External communicators suffer from many of the problems described above but in addition the character length of variables is restricted further due to external constraints. In some cases the external communicator is not implemented.

The ABSOLUTE communicator is always machine dependent.

- The character string is partially defined in CORAL 66 and an assignment of the form :

```
s := "A character String";
```

assigns a location to the integer mode (or type) variable. The location assigned is often different in differing implementations. For example the ICL George 3 implementation assigns a 2 word block to each character string. The first word is a pointer to the starting byte of the string whilst the second word is the length of the string. In the Intel 8080 version of the CORAL 66 compiler the location of the string is the first byte of the string which contains the length of the string (which is thereby restricted to 256 characters).

There are no string manipulation, eg substringing, constructs defined in CORAL 66.

- The CORAL 66 language assumes implicitly word addressing, rather than byte. For example the declaration :

```
INTEGER i,j,k;
```

declares 3 integers with consecutive addresses such that indirect references [LOCATION(i)+1] references the variable j, and [LOCATION(i)+2] references k. Many implementations introduce the concept of BYTE but do not allow indirect references as shown above.

- Bit slicing of integers is permitted. If programs use these constructs, transfer of programs to a machine with a smaller word size can be difficult.

- Code inserts are permitted. The form of the code insert is left to the implementation.

- No input/output is defined and implementations define procedures to perform the functions. Procedures tend to be different for different implementations.

- No process communication or syncronisation facilities are provided for processes operating concurrently. Many programs resort to code inserts to access operating system facilities to provide these functions.

2.2.2 Universal Assemblers

The use of a general assembler language which may be mapped, by suitable means, to any reasonable assembler language of a real machine has been used as a mechanism to achieve tranportability. The general features of this method are discussed in [Reference 3 : Brown (Editor)] whilst in the implementation of STAB-12 [Reference 10 : Colin 1975] the intermediate language (see sections 2.2.3 and 2.3.1) is a low level, assembler like, language. The assembler languages are generally of a low level and can often be interpreted by using a macroprocessor (see section 2.3.2). The low level means that the effort required to provide an interpretation system on a new computer may be significantly less than the effort required to ether rewrite the program for the new machine, or provide a compiler to support a high level language. The disadvantage is that the effort required to produce the program initially may be high. In addition the low level nature of assembler languages and lack of structure may mean that the programs are difficult to understand and modify subsequently.

2.2.3 Abstract Machines

An abstract machine is one defined with an architecture which reflects that of a number of real machines. Abstract machine basic features are discussed in [Reference 3 : Brown (Editor)], and [Reference 5 : Warren 1976]. The BCPL compiler implementation [Reference 6 : Richards 1971] utilises a simple stack based abstract machine, as does the Mobile programming system [Reference 7 : Coleman, Pool & Waite 1974]. Most portable compiler systems incorporate an abstract machine architecture within the compilation system. This is discussed further in section 2.3.1.

Some abstract machines [Reference 13 : Poole 1971] are multi-level with one level defined upon the level beneath.

There are various types of abstract machines defined:

For example -

- A simple 2-register machine where the registers are used to store values of operands. Operations are defined on these operands, eg COMPARE, ADD, MULTIPLY,.. etc.

  These machines require storage for data objects and operations to allow flow control both conditionally and unconditionally. In addition operations are required to allow data transfer between registers and storage.

- An extension to the 2-register machine is one with an finite or infinite number of virtual registers.

  The variations on this theme depend upon the overall

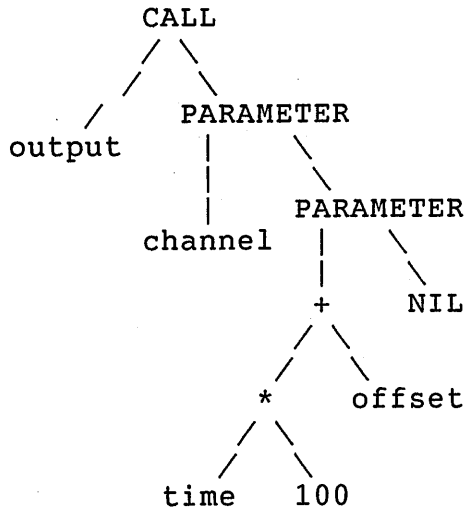
architecture of the abstract machine, eg all operations could act upon the contents of the virtual registers, including flow control operations. If this were the case the virtual registers would contain references to both data and program storage.

Conversely the virtual registers could be used in exactly the same way as the 2-register machine uses its registers. In this case the operations required would be similar to those of the 2-register machine except that they would need to reference source and sink virtual registers for the operation.

- A tree structured machine where the structure of the program is represented by a linked structure. The procedure call

```
output(channel,time*100+offset)
```

may be represented by the tree :



The tree structure shown here is just one method of portraying the procedure call and has omitted operand identification nodes, eg a LITERAL node to show that the number "100" is a literal, or an OPERAND node to show that "time" is an operand.

This kind of abstract machine is interpreted by traversing the tree in an ordered manner.

- A simple finite state machine where the current state of the machine and the value of an input determines the next state, and any output.

This can be extended to include a stack. In this case the state that the machine takes is dependent upon the current state, the value of the item on the stack and the value of an input.

Abstract machines can be used as vehicles for transporting programs. They may be modelled as in [Reference 20 : Newey, Poole and Waite 1972] but usually the transportation of programs requires the realisation of the abstract machine in the new environment. This can be by :

- Providing a simulator for the abstract machine in the new environment, a technique used in the first stage transportation of BCPL compilers [Reference 6 : Richards 1971] and in language interpretation systems for LISP [Reference 25 : Taylor 1977] and Pascal [Reference 29 : Ammann 1977].

- Translating the "object code" of the abstract machine into the object code of the new environment. This would allow the program to run directly in the new environment. This is the favoured method for implementing efficiently compilers on new machines as in [Reference 8 : Capon, Wilson, Morris and Rohl 1972] and [Reference 27 : Wilcox 1971].

Abstract machines, as a method of achieving program portability, are often used in conjunction with a compiler and this will be discussed further in section 2.3.1.

2.3 Tools

This section looks at software tools which provide for program transportability.

2.3.1 Compilers

The name compiler is given to a program which will translate from a high level language into an object code which can be directly executed on a target machine.

So if a program is written in a high level language and a compiler is available for that high level language on all required machines then the program is portable to all of those machines (subject to the constraints described in section 2.2.1).

It is desirable, therefore, to make the compilers themselves portable. Techniques for making portable compilers, and using compiler-compilers, or syntax generators, are well documented.

The most popular technique is the use of a 2-stage compilation process. Such a method is used in implementing compilers for BCPL [Reference 6 : Richards 1971], C [Reference 9 : Snyder 1975], Pascal [Reference 29 : Ammann 1977] and CORAL 66 [Reference 4]. The first stage is the lexical and syntactic parsing, and syntax/semantic checking to the rules applied by the language definition. The output from this stage is an intermediate language representation which is logically identical to the original program .

The intermediate representation is often the object code of an abstract machine which is defined to fit the language being implemented. There have been attempts to make the abstract machine universal, ie applicable to all languages and all real machines. None have been truly successful. This is often termed the Uncol problem as the intermediate language, Uncol, was aimed at being universal. Uncol was defined by Steel in 1961 [Reference 22 : Steel 1961]. The problem to be solved is described in terms of 'n' languages requiring implementation on 'm' target machines. If a 2-stage compilation process is used then 'n*m' phases would need to be implemented to provide all languages on all machines. If any code generating phase for any specific target machine can be used with all source languages then only 'n+m' phases would need to be implemented. The major problem of this approach is that the interediate language needs to cater for the requirements of all (or at least a wide range) of languages and map onto the real machine architectures of target machines. In addition the intermedate language would need to be flexible enough to take advantages of the evolution of future languages and future machine architectures. These problems are described in [References 12 and 24 : Koster 1974].

The second stage of the compilation process involves the translation of the intermediate representation into the object

code of a real machine, the target machine.

To transport a program it is necessary to have a second stage capable of translating the intermediate representation of the program to the desired target machine.

The machine on which the compiler executes is the host machine, to avoid confusion with the target machine which was defined above. It is not necessary for the host and target machines to be the same. If the compiler generates object code for a machine which is not the host it is called a cross compiler.

A cross compiler which is, itself, written in its source language, ie the language that the compiler translates, can be made to operate on its target machine (and thereby ceasing to be a cross compiler) by exploiting a process known as bootstrapping. Bootstrapping methods, with respect to the SNOBOL 4 language, are discussed in [Reference 21 : Dunn 1973].

The sequence of events is :

- the source code of the compiler is input to the operating compiler on the host machine.

  This generates, firstly, an intermediate representation and then the target object code representation.

- the target object code is transferred to the target machine and the code run. The target machine then becomes the host machine for the compiler which will generate object code for the new host machine.

2.3.2 Macroprocessors

The macroprocessor has been used as a successful tool in program transportability. Macroprocessor basics are discussed in [Reference 3 : Brown (Editor)] and examples of their use appear in [Reference 13 : Poole 1971].

A macroprocessor is a general purpose tool which performs text processing and parameter substitution functions. It relies upon 2 stages, namely macro definition and macro expansion.

For example a macro definition may be :

```
MACRO &language-construct
     CASE &language-construct IS
       call: {LINE proc&1}
       goto: {LINE goto LINE&1}
     END-CASE
END-MACRO;
```

In the above example variables used in the macroprocessing stage

are preceded by the "&" character and "&1" is used to denote actual parameter 1 to the macro expansion. Macroprocessor keywords are shown in capital letters.

The example is of a fictitious macroprocessor input language and is provided for example only. The macroprocessor recognises the example as a macro definition. The macro is given the name "&language-construct" which, in this example, can be either "call" or "goto". The action taken, on expansion of the macro, will be dependent upon which of these names is used.

The macroprocessor will use the above definition in macro expansions, so :

```
call input ;
goto exit-label;
```

Will be expanded into :

```
10 procinput
20 goto 750
```

To translate source code which is in one language into another language it is necessary to define macros for the source language which, when expanded, will generate the target language. This method can be used to transport programs from one machine to another where the same high level language is not available on the host and target machine. The program is translated by the macroprocessor, using macro definitions produced specifically to enable the transportation, from the high level language of the host to that of the target.

Where the two languages are significantly different it is possible to perform staged translation. This involves generating an intermediate representation which is further processed to generate either a further intermediate version (and again and again if necessary), or the final target code.

The macroprocessor is a very flexible portability tool. Its value depends upon the richness of the macro definition language and its speed of execution.

2.3.3 Interpreters and Emulators

An interpreter is a program which executes the instructions of an abstract machine, whilst the emulator executes the instructions of either a real machine or an abstract machine. Interpretation is a common method of implementing compilation systems quickly. The interpreter executes the instructiions of the intermediate language. Such techniques are used in implementing portable compilation systems for BCPL [Reference 6 : Richards 1971], Mobile programming system [Reference 7 : Coleman, Poole, Waite 1974], C [Reference 9 : Snyder 1975], STAB-12 [Reference 10 :

Colin 1975], L-Star [Reference 23], Lisp [Reference 25 : Taylor 1977].

A program running on a machine relies, for its operation, upon something executing the instructions which make up the logic of the program. The possibilities are :

- The program is in the executable code of the host machine and the code is interpreted by the hardware of the host machine. It may be that the executable code is generated as a result of compilation.

- The program is in some intermediate form, eg the code of an abstract machine, and this is being interpreted by a program called an interpreter. The language BASIC often operates in this manner as does the PASCAL P-code compiler.

Transportation of the program can be accomplished, respectively, by :

- Implementing on the target machine an emulator which provides, in software, the ability to execute the code of the host machine.

- Implement the abstract machine interpreter on the target machine.

In practice the implementation of an abstract machine interpreter is much simpler than implementing the instructions of a real (and sometimes alien) machine and it can be simpler than implementing a compiler for the target machine.

Interpretation can be used to aid program transportability either as a primary aid or in conjunction with a compiler and abstract machine interface. Interpreters may be slow in execution.

2.3.4 Total Environment

One method of ensuring that a program is completely portable is to implement it in an environment which provides all of the facilities that the program needs and bar access to facilities which are provided outside of the environment.

To port the program to another identical environment should pose no problems.

The problem with this approach is that the environment itself needs to be portable. To overcome this the environment is built from a small kernel of facilities, say f1. f1 facilities are produced using the host machine facilities, eg PASCAL and Operating System calls. This kernel will need rewriting if the environment is moved to another machine.

The environment is enhanced, say to f2 facilities using f1 facilities alone.

Again the environment is enhanced, to f3 facilities, but this time using f1+f2 facilities, ... and so on.

Each level of facility increase relies upon the facility level below.

It is clear that the initial kernel must be both small for minimum rewrite on porting but be large enough to provide for f2+ facilities without their recourse to using host operating system facilities.

3 AN INTERMEDIATE LANGUAGE

This section of the report describes an intermediate language designed for a High Level Language Compiler. The intermediate language could be used as the interface between a compiler and :

- a number of code generators which would provide the ability to translate a high level language into the code of a range of computers.

- an optimiser which would act upon the intermediate code and transform it such that the code generator would generate more optimal code, minimising either the run time of the program, or the memory utilisation.

- an interpreter to permit the execution of the intermediate code. The interpreter would act upon the intermediate code and obey the instructions of the code giving the effect of the program executing.

The intermediate language, which for ease of reference is called Ivor in this document, is the code of an abstract machine. The abstract machine and the Ivor language will be described in this section.

The reason for designing Ivor was to assess the difficulty of mapping modern programming structures into some intermediate form and to assess the likely problems of generating code for a range of target machines.

The design of Ivor was heavily influenced by the Chill and Ada programming languages. Intermediate languages like O-code for BCPL, P-code for PASCAL and Janus did not address the problems associated with program segmentation and of programs operating concurrently.

All practical compilation systems require programs to be separated into manageable units, compiled separately and linked to form executable program object code. In order to cater for this a compiler needs knowledge about items used within a program segment but which are extenal to it. No intermediate language which was investigated during this research period provided for program segmentation and separate compilation.

In addition to this, many compilers implement a particular storage strategy and this is mapped, implicitly, into the intermediate language structure. For example, a Fortran COMMON approach is often adopted for external data. Data is addressed via an offset from some data area and is positionally dependent within the COMMON area. Any modifications to a program's external data would inevitably mean the recompilation of all program segments. Data within procedures are often declared with reference to a stack, or stacks. Both of the storage allocation strategies are inherent in the design of the intermediate language.

Concurrent processing primitives were a recent innovation at the time of the research study and as a consequence did not feature in any of the intermediate languages considered. Concurrent processing facilities of the high level languages were developing also and, as a result, it was not possible to adopt a single strategy for handling concurrency. Two were chosen, the signal and buffer (as in Chill) but a third type, the Event, would probably be needed in order to cater for Ada.

In addition to the primary objectives stated above, secondary design objectives were based upon:

- The lack of optimisation capabilities in intermediate languages. For example A:=A+B can generate intermediate code of the form :

```
STACK A
STACK B
ADD
STORE A
```

A non-intelligent code generator will not appreciate that the variable 'A' appears on both sides of the assignment until too late. A better approach would be to give the code generator the maximum amount of information in a single operation, ie

```
add B to A result_in A
```

Also, labels are often generated during intermediate language production. Optimisation of code across labels is very difficult as execution can be transferred directly to the label from an unknown program state. Path analysis, using Graph Theory (as described by Wilson in reference 40) can be used, but the algorithms are complex and require considerable processing time. A way of avoiding the problem is to mark some labels as special, having a small number of places from which execution can be transferred. The state analysis for this case is much simpler.

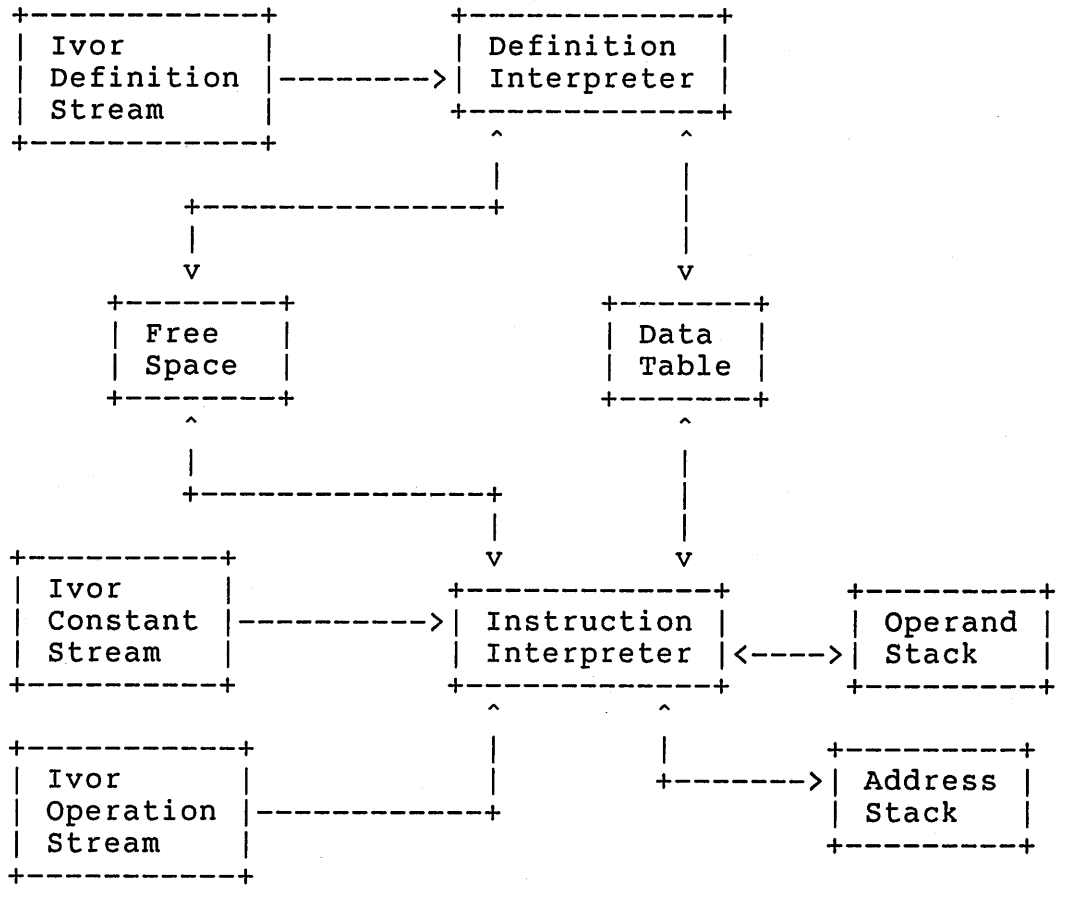
- The lack of source related information to allow symbolic debugging at interpretation or final execution stage is apparent in the intermediate languages considered. It was thought desirable to preserve source names and modes in Ivor to facilitate high level debugging. Ivor did not go far enough in this. Source line, and probably source column, references would have been necessary to allow sensible symbolic debugging.

The conclusion reached from performing this design exercise is that the higher the number of basic concepts (or operations) in an intermediate language the more difficult is the task of generating code, though the task of implementing each construct may be simpler if the intermediate language has simple operations

defined.

The design described here uses :
- Simple linear operations
- Stacks and stack operations
- Flow control constructions
- Process syncronisation constructions

3.1 Ivor Abstract Machine



The Ivor abstract machine has the components shown above. To cater for program segmentation the Ivor program is divided into units. Units are sub-divided into streams. These are the Definition, Constant and Operation streams.

The definition stream introduces objects (more on these later) and each object has an identification key. Objects are, for example simple integers, arrays ..etc.

The constant stream introduces constant expressions, for example (3+7/16) which need to be evaluated. The constant stream is needed if the Ivor interface is to be used by a code generator which is generating code for, say, a different word length machine than that hosting the compiler. If it is necessary to evaluate the constant expressions in the target machine environment then this can be done.

The operation stream is the code of the abstract machine which governs execution, data manipulation and concurrent processing.

The abstract machine has 4 separate data areas: Free Space, the Data Table, the Operand Stack and the Address Stack.

Free space, and the 2 stacks are used to hold the values of objects whilst the data table holds information about objects.

To ensure that the structure of Ivor is free from some storage allocation strategy the structure of free space is left undefined, except that free space is not directly addressable and must be addressed through the address stack, in the case of a calculated address, or the data table for an introduced object. The mode of any object held in free space is an integral part of its address.

The 2 stacks are used for holding, temporarily, the results of evaluated expressions either in address calculation, in which case the address stack is used, or in other expressions, in which case the operand stack is used.

The data table is used to hold information about objects introduced, including the objects free space address and its structure (more on this later). All objects are identified by a key, the identification key, which is used to index the data table.

The operation of the abstract machine is :

- The instruction interpreter selects the first Ivor unit for reading and starts with the constant stream.

- The constant stream of that unit is read and processed, operation by operation, by the instruction interpreter. The processing causes the evaluation of each constant expression placing the result in the data table. Future reference to the constant expression, ie in the definition or instruction stream, is by an identification key which indexes the data table.

- The definition interpreter reads and processes the Ivor definition stream. In doing this the definition interpreter reserves the right amount of free space and puts a free space address in the data table of the object introduced. The data table record for each object is created from information provided in the definition stream.

- The operation stream is then read and each operation executed. Generally the Ivor instructions are executed sequentially. However some instructions provide for the execution of particular parts of the Ivor program, eg by branching to a procedure call. The valid operations will be described later.

The description of Ivor which follows will use the abstract machine to describe the operation of Ivor's functions. The description will cover :

- Program segmentation
- The definition stream
- The operation stream
- The constant stream

Ivor will be described using a modified Backus-Naur form of syntax description method. The semantics are described informally in English.

Syntactic categories are referenced by one or more words enclosed between angle brackets (ie <.....>) and is termed a non-terminal symbol. The non-terminal symbol may contain comments, for explanatory purposes, in which case the comment is underlined, eg <procedure operand>. The comments do not form part of the syntax description and in the above example the non-terminal symbol, <operand> is exactly equivalent to <procedure operand>.

Grouping of syntactic elements is by the use of braces (ie {....}). The group may be repeated by specifying an upper and a lower bound immediately following the braces, eg {<procedure operand>}1:10 indicates that the non-terminal symbol must appear once, and may be repeated upto 10 times. An indefinite upper limit is shown by an asterix (*) as the upper bound. The special case of a symbol being optional, eg {<procedure operand>}0:1, may be represented using square brackets, eg [<procedure operand>].

Terminal symbols are shown, where appropriate, in capital letters. Although the intermediate language is shown with terminal symbols represented by English words it is expected that the language will be compactly coded and not directly human readable.

Other meta language constructs are as the standard Backus-Naur and are described only briefly here.

| Denotes an alternative.

::= Means "defines".

3.2 Program Segmentation Definitions

```
<program> ::=
        IVOR <system name>
        { UNIT <unit data>
```

```
        <definition stream> <constant stream>
        <operation stream> ENDUNIT}0:*

<system name> ::=
     <character string>

<unit data> ::=
     <module name> {<seized data>}0:*

<module name> ::=
     <character string>

<seized data> ::=
     SEIZED <module name> <definitions> ENDSEIZED

<character string> ::=
     CHARSTRING <character> ENDCHARSTRING

<character> ::=
     CHARACTER <internal code of character number>
```

As has been said in the preceding section an Ivor program consists of one or more units, each of which comprises the 3 streams, definition, constant and operation. One of the requirements arising from the need for program segmentation is that each unit needs to be completely separate from others and needs to be self contained so that the Ivor abstract machine can take unambiguaous action for every operation.

The unit is part of a system which is specified by a system name. The abstract machine will not permit the concurrent operation of units with dissimilar system names.

Each unit is identified by a module name. The abstract machine requires each unit of a system to have a unique module name.

In order to satisfy the requirements of separate compilation, where a unit uses objects which are introduced in a different unit, eg a procedure is called in module "A" which is defined in module "B", it is required that the object be introduced as a "seized" object in the module using (but not defining) it. In the module in which the object is introduced it is required that the object be granted wider access, see section 3.3.

A seized object is introduced in exactly the same manner as one which is introduced in the definition stream, except that it may not be marked as "granted". To allow a compiler to provide simply the information required seized objects are grouped so that all of the objects seized from a particular unit appear in the same seized data group in the unit. The name of the unit from which the objects are seized is given by the module name of the seized data. For example :

- A unit seizes objects "A","B" and "C" from a unit with module name "M" and object "X" and "Y" from a unit with module name "N".

- The unit which seizes "A","B","C","X" and "Y" will contain 2 seized data entries. The first will specify module "M" and introduce the objects "A","B" and "C" whilst the second will specify module "N" and introduce object "X" and "Y".

The seized data may contain introductions of objects which are not used in the unit in addition to those which are. It is not required that each unit of a system have identical seized data and introductions may be in any order in each unit. Seized data is not like Fortran "common" data in that ordering is not important. The abstract machine will link between units symbolically, using the module name and the object's identification key.

3.3 Definition stream

Definitions :

```
<definition stream> ::=
     DEFINITIONS <definitions> ENDDEFINITIONS

<definitions> ::=
     {ABSTRACTION <abstraction data> <declarations>
      ENDABSTRACTION}0:*

<abstraction data> ::=
     <visibility> <nesting> <abstraction descriptor> <new
     key>

<abstraction descriptor> ::=
     PROCEDURE <procedure data>
     | INNERPROCESS | OUTERPROCESS

<procedure data> ::=
     {INLINE | GENERAL | SIMPLE} { RECURSIVE | NONRECURSIVE}
     {CRITICAL|NONCRITICAL}

<visibility> ::=
     GRANTED | BUILTIN | VISIBLE

<nesting> ::=
     <life time defining key> <visibility defining key>

<new key> ::=
     <character string> <key>

<key> ::=
     KEY <number>
```

The definition stream introduces Ivor identification keys which

represent objects used later in Ivor operations. An object does not exist in isolation but exists only when an abstraction is activated. The object is said to be tied to that abtraction.

Ivor abstractions are procedures and processes with a special type of process, the outer process, for which an instance is always active when the system is active.

Processes can operate concurrently whilst procedures cannot. Many instances of the same process can be active at the same time. The abstractions are invoked, or activated, by Ivor operations in the operation stream. Both procedures and processes may have parameters, or return results as mechanisms for passing arguments between invoking and invoked environments. The parameters and results are defined in the <declarations> immediately following the abstraction introduction.

Both Ada and Chill allow operations on procedure mode objects, eg

```
p := proc_name;
.....
.....
p();
```

Indirect calling of procedures in this way causes inefficiencies in the run-time code, especially if all procedures are candidates for indirect calls, even though some may never be called in that manner. In Ivor procedures can be called indirectly only if they are introduced with appropriate attributes, see below.

To allow a compiler to indicate that some procedures can be optimised (though they need not be if the code generator is non-intelligent) special attributes are provided. Misuse of these attributes by a compiler is likely to cause incorrect code to be produced.

INLINE - The procedure code will be expanded in-line at the point of call.

GENERAL- The procedure may be used in an assignment operation and may be invoked indirectly.

SIMPLE - The procedure may not be used in an assignment operation.

RECURSIVE The procedure may invoke itself either directly, through another procedure or indirectly.

NONRECURSIVE
- The procedure may not invoke itself in any way.

CRITICAL The procedure may be invoked only when no other

procedure contained in a particular bracketed area of the operation stream, termed a critical region, is active.

NONCRITICAL
- A normal, non-critical procedure.

So that many storage allocation strategies are possible, abstractions must not be nested in the Ivor definition stream. The static nesting characteristics are given by specific nesting attributes.

In addition, an object has a lifetime and a visibility. The lifetime of an object is the period during which storage must be allocated to the object. An object starts its life when its tied abstraction is activated. Global objects are tied to the outer process. An object can be living but not visible and visibility is separately definable. This would cater for nested declarations of the form :

```
BEGIN
DECLARE x INTEGER;
   ...
   ...
        BEGIN
        DECLARE x BOOLEAN;  -- Outer x is not visible
        ...                 -- but it is alive
        END;
   ...
   ...
END;
```

For example :

The four abstractions p1,p2,p3 and p4 are defined informally as follows :

p1 is the outer process.

p2 is an inner process which is tied to p1.

p3 is a procedure which is tied to p1 but is visible only when p2 is active.

p4 is a process which is tied to p2 and is visible whenever p1 is active.

The Ivor intermediate language caters for these cases by providing for a lifetime defining key and visibility defining key to be specified as the nesting attribute.

The key specifies either a tied abstraction, in the case of a lifetime defining key, or a bracket or abstraction name, for a visibility defining key.

3.4 Declarations

Definitions :

```
<declarations> ::=
     {<introduction>}0:*

<introduction> ::=
     <object introduction>
    |<mode introduction>
    |<synonym introduction>
    |<notion introduction>
```

Ivor declarations are used to introduce objects, modes, synonyms and notions which are described further in the following sections.

ALL identification keys used within the operation or constant streams must be introduced. The only exception to this is the temporary object of section 3.5.2.8.

3.4.1 Objects

Definitions :

```
<object introduction> ::=
     OBJECT  [ <object introduction details> ]
     {<new object key>}1:* ENDOBJECT


<object introduction details> ::=
     <visibility> <nesting> <initialisation> <accessibility>
     <mode key>

<initialisation> ::=
     LIFEINIT | REACHINIT | NOTINIT

<accessibility> ::=
     DIRECT
    |ARGUMENT <abstraction key> <argument type>


<argument type> ::=
     IN | OUT | INOUT | LOC | RESULT
```

Ivor objects occupy free space and the definition interpreter reserves an appropriate amount of free space for the object and places a free space address for the object in the data table record which is referenced by the object's identification key.

More than one object may occupy the same free space locations provided that their tied abstractions cannot be simultaneously active.

Objects may be initialised by operations in the operation stream. They may be initialised statically (ie once only when the outer process is activated) or may be initialised every time that an instance of the tied abstraction is activated. LIFEINIT is the former case whilst REACHINIT is the latter.

Parameters to abstractions are introduced immediately following the ABSTRACTION introduction as objects. They are identified as having an accessibility attribute of ARGUMENT. The identification key of the abstraction is included as an attribute. The argument types supported by Ivor are :

IN — The value of the actual parameter of the procedure/process is copied to local storage at the procedure/process invocation. The local storage is used throughout the procedure/process execution and the original actual parameter value is unchanged at the procedure/process exit.

INOUT — Identical to the IN argument except that at procedure/process exit the value of the parameter is copied to the location represented by the actual parameter.

OUT — Identical to the INOUT except that the value of the actual parameter will not be used within the procedure/process.

LOC — The actual parameter location will be accessed for all read and write references to the parameter within the procedure/process.

RESULT — The argument will be used to hold the result (eg in the case of a function call) of a procedure/process.

The structure of an object is given by its mode which specifies attributes such as whether the object is an array, or simple and whether it can be used to hold integers, strings, etc. The abstract machine uses the mode information to determine how much free space is required and to determine the kinds of operations which are allowable on the objects. The mode information is obtained from a data table entry which is referenced by the mode identification key. Modes are described next.

3.4.2 Modes

Definitions :

```
<mode introduction> ::=
     MODE  [ <mode introduction details> ] <new mode key>
```

```
     ENDMODE

<mode introduction details> ::=
     <visibility> <nesting> <capability> <mode construction>

<capability> ::=
     R | W | RW

<mode construction> ::=
     <basic mode>
    | REF <mode key>
    | BUFFER <mode key>
          <buffer length constant>
    | <array mode>
    | <structure mode>

<basic mode> ::=
     BIT | FLAG | CHAR | LABEL | PTR |
     BIT | FLAG | CHAR | LABEL | PTR |
     PAIR | CASELABEL | RANGE <range> |
     PROC | INSTANCE

<range> ::=
     <lower bound constant>
     <upper bound constant>

<array mode> ::=
     ARRAY <mode key> <range>

<structure mode> ::=
     STRUCTURE { <field> }1:* ENDSTRUCTURE

<field> ::=
     <mode key> <new field key>

<constant> ::=
     NUMBER <number>

<number> ::=
     { 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F }4:8
```

The mode of an object defines the structure of the object and the range of operations which may be performed on the object. In addition it defines whether the object may be read from and/or written to.

The capability attribute of the object specifies R for read only access, W for write only and RW for read/write access.

The mode of an object is one of : basic, reference, buffer, array or structure.

3.4.2.1 Basic Modes

Objects which are defined to be a basic mode are simple and can hold a value which is restricted to one valid for the basic mode.

Basic modes are :

- BIT is a data item which can take the value TRUE or FALSE.

- FLAG is a data item which can take the value FREE or BUSY.

- CHAR can take the value of the internal code for any character which can be displayed and printed.

- LABEL is a reference to a point in the operation stream of the Ivor program.

- PTR is a free space reference to any object.

- PAIR is a pair of integers.

- CASELABEL is a reference to a point in the operation stream of an Ivor program which can be entered only by a CASE operation.

- RANGE is an integer which can take a value which is >= the lower bound specified and <= the upper bound.

- PROC is a reference to a point in the operation stream of an Ivor program which represents the start of a GENERAL procedure.

- INSTANCE is an identity given to a process when it is executing. A process may be started by one or more process start operations and each instance of the process can execute concurrently. Each instance of the process is given a unique process instance value by the Ivor abstract machine and this may be assigned to an object of mode INSTANCE.

3.4.2.2 Reference Modes

A reference mode is a free space reference to an object which has the mode specified by the mode identification key.

3.4.2.3 Buffer Modes

A buffer mode object is used for communicating values between concurrent processes. The buffer has a mode specifed by the mode identification key and has a length specified by the constant.

3.4.2.4 Array Modes

An array mode is an ordered set of elements each of identical mode. The mode of each element is given by the mode identification key, which may, itself, be an array mode. The number of elements is given by the difference between an upper

and lower bound (plus 1). An array element is referenced by an index value which must be >= the lower bound and <= the upper bound.

3.4.2.5 Structure Modes

A structure mode is a collection of elements, called fields, each of which may be a different mode. The field mode is given by the field mode identification key.

3.4.3 Notions

Definitions :

```
<notion introduction> ::=
     NOTION <notion introduction details> <new key>
          ENDNOTION

<notion introduction details> ::=
     <visibility> <nesting> <notion details>

<notion details> ::=
     LABEL { STRUCTURED | UNSTRUCTURED }
   | BRACKET  <bracket type>
   | SIGNAL

<bracket type> ::=
     { STRUCTURED | UNSTRUCTURED }
   { CRITICAL | NONCRITICAL }
```

Notion introductions introduce label, bracket and signal type identification keys.

Label notions are used to mark points in the Ivor operation stream where flow is transferred to directly, rather than sequentially from the preceding operation. It is useful for a code generator to know which labels (and brackets) are branched to from a few, rather than many, places. This makes the job of code optimisation simpler as the number of possible states available at the label are fewer. Labels with few branches are termed structured labels. Structured labels are usually generated as a result of some high level language loop, or conditional statement. An unstructured label may be branched to from many places, or may be assigned to a label mode object.

A bracket notion is used to delimit a section of Ivor program which is to be entered at its head and where an exit from the bracket will transfer control to the operation following the bracket. Brackets cater also for concurrent operation of critical regions. If data which is shared between concurrently operating processes is to be accessed it is desirable to prevent concurrent (write) access to the data. this can be achieved by specific programming which ensures that data accesses occur only within a "critical region". If the bracket is marked as critical only one process instance may have control dynamically contained within the bracket at any time.

Signal notions are used in the syncronisation between concurrently operating processes.

3.5 Operation Stream

```
Definitions :
     <operation stream> ::=
          OPERATIONS { <sequential operation> |
          <non sequential operation> |
          <control operation> }0:* ENDOPERATIONS

     <operand> ::=
          {  <identification  key>  | <stack operand>  |
          <literal operand> } <operand descriptor>

     <stack operand> ::=
          { OPERANDSTACK | ADDRESSSTACK } <mode key>


     <literal operand> ::=
          <literal> <mode key>

     <literal> ::=
          <constant> | <character> | <character string> |
          NULL | FREE | BUSY | TRUE | FALSE | UNDEF | NONE

     <operand descriptor> ::=
          VALUE | CONSTANT
```

The operation stream introduces the operations which are to be executed by the Ivor instruction interpreter to provide the logic of the program. Operations are either sequential, non sequential or control operations. Sequential operations are interpeted one following the next. Non sequential operations transfer control to particular Ivor instructions within the operation stream.

Operations usually have one or more operands. The operands provide the values and addresses used in the operations.

An operand can be one referenced by an identification key, in which case the information about the object is obtained from the abstract machine data table. Alternatively the object may be a stack reference. If a stack reference is specified, either the address or operand stack, its mode is given by the mode key. The mode information is stored in the abstract machine data table record referenced by the identification key. An object may be a literal and can be any of the appropriate literal types.

The operand descriptor gives the abstract machine information about how to use the operand, ie :

- If the operand descriptor is VALUE the operand is used as an address and the value of the operand obtained from it by indirection. For example suppose the operand is :

  OPERANDSTACK VALUE

  The value from the top of the stack is used as the operand, ie the operand OPERANDSTACK is used as the address of the

top of the stack and indirection used to obtain a value.

Conversely, suppose the operand is :

KEY x VALUE

In this case the value is obtained from a free space address given in the data table record of the identification key x and allocated to x.

The VALUE operand descriptor may not be used with literal operands.

- If the operand descriptor is CONSTANT no indirection is applied and the actual value or address is used. If the above examples are repeated with CONSTANT as an operand descriptor the address of the stack top, and free space address of the key x, respectively, will be used as operands.

The operations will be defined with respect to the abstract machine and the following notation is used within the description.

Firstly the syntactic definition is given, in exactly the same manner as described under th definition stream, above.

Secondly a list of valid modes and notions are given for each operand. The list describes the combination of modes which are valid. Any other combinations are invalid and the abstract machine will generate an error condition if an invalid combination of modes or notions are provided.

The description of modes and notions follow the following format: Firstly a "Modes" heading to indicate that the modes and notions are being described. This is followed by a heading indicating which part of the syntax description is being categorised, for example, suppose the syntax definition was :

```
Definitions :
     <relational operation> ::=
          { GT | GE | LT | LE | EQ | NE } <result operand>

          <first source operand> <second source operand>
```

A Mode or notion section might be :

```
Modes :
     For <relational operation> is GT
     Order is <first source operand> <second source operand>
          <result operand>

     BIT        BIT        BIT
     CHAR       CHAR       BIT
     RANGE      RANGE      BIT
```

The meaning of this is is :

- If the relational operation is the GT operation then the first operand may be a BIT mode in which case the second operand must be a BIT mode which will generate a BIT mode result operand, and so on for CHAR and RANGE mode first operands.

- The "Order is .." part of the definition gives the order of the modes with respect to the syntax definition.

```
     For <relational operation> is GT | GE | LT | LE | EQ | NE
     Order is <first source operand> <second source operand>
          <result operand>
     BIT        BIT        BIT
     CHAR       CHAR       BIT
     RANGE      RANGE      BIT
```

The above defines the modes for all of the <relational operation> options and the following shorthand notation may be used when the mode definition applies to all options.

```
     For <relational operation>
     Order is <first source operand> <second source operand>
          <result operand>
     BIT        BIT        BIT
     CHAR       CHAR       BIT
     RANGE      RANGE      BIT
```

The actual functions performed by the Ivor operations are described with respect to a smaller number of pseudo operations of the abstract machine, eg

```
Function :
     For <relational operation> is  GT :
          if value(<first source operand>) >
```

```
        value(<second  source operand>) then
    <result operand> := TRUE else
    <result operand> := FALSE
    endif
```

The meaning of this is : If the relational operation is GT then the abstract machine executes the pseudo operations shown, in this case if .. then ... else .. endif and value pseudo operations. Pseudo operations are explained under a separate heading at their first occurance. In this way the operations of Ivor can be explained with reference to a smaller number of basic constructs.

Where necessary, additional explanation will be given in order to relate the Ivor constructs to high level programming language features.

3.5.1 Sequential Operations

Definitions :

```
     <sequential operation> ::=
          <relational operation> |
          <arithmetic operation> |
          <logical operation> |
          <numerical operation> |
          <complex mode accessing operation> |
          <assignment operation> |
          <result operation> |
          <syncronisation operation> |
```

Sequential operations cater for source language expressions, eg arithmetic and logical expressions and array and structure accessing. In addition Ivor operations are provided for the mapping of returned values from procedures and processes.

Some bit manipulation operations are provided also.

3.5.1.1 Relational operation

Relational operations allow two operands to be compared, ie using <,>,<=,>=,/= and = operations, producing a boolean result.

Definitions :

```
     <relational operation> ::=
          { GT | GE | LT | LE | EQ | NE } <result operand>
          <first source operand> <second source operand>
```

Modes :

```
     For <relational operation> :
     Order is <first source operand> <second source operand>
          <result operand>
     BIT       BIT       BIT
     CHAR      CHAR      BIT
```

```
     RANGE     RANGE     BIT

Function :
     For <relational operation> is  GT
          if value(<first source operand>) >
             value(<second  source operand>) then
          <result operand> := TRUE else
          <result operand> := FALSE
          endif

     For <relational operator> is  GE
          if value(<first source operand>) >=
             value(<second  source operand>) then
          <result operand> := TRUE else
          <result operand> := FALSE
          endif

     For <relational operator> is  LT
          if value(<first source operand>) <
             value(<second  source operand>) then
          <result operand> := TRUE else
          <result operand> := FALSE
          endif

     For <relational operator> is  LE
          if value(<first source operand>) <=
             value(<second  source operand>) then
          <result operand> := TRUE else
          <result operand> := FALSE
          endif

     For <relational operator> is EQ
         if value(<first source operand>) =
             value(<second  source operand>) then
          <result operand> := TRUE else
          <result operand> := FALSE
          endif

     For <relational operator> is  NE
          if value(<first source operand>) /=
             value(<second  source operand>) then
          <result operand> := TRUE else
          <result operand> := FALSE
          endif
```

Pseudo operations :

value The reference level of operands is specified explicitly in Ivor operands, see start of section 3.5. If B is an operand with VALUE specified as an operand descriptor then (value(B)) is the value of the operand obtained from the free space address of the operand. If B is an operand with CONSTANT specified then (value(B)) is the value

of the constant - which may be a free space address.

if — The (if .. then ... else .. endif) operation and associated relational operations have the usual interpretation, ie suppose A and B are operands then (if A>B then C else D endif) will cause C to be executed if (A>B) or D to be executed otherwise.

3.5.1.2 Arithmetic Operations

Arithmetic operations allow the common programming language operations of addition, subtraction, multiplication, division, modulo, absolute and remainder to be represented in Ivor. Also provided for are successor and predecessor functions. These are defined specifically for integer, character and boolean type operands.

Definitions :

```
<arithmetic operation> ::=
     <monadic arithmetic operation> |
     <diadic  arithmetic operation>

<monadic arithmetic operation> ::=
     { SUCC | PRED | MINUS | ABSOLUTE } <result operand>
     <source operand>

<diadic  arithmetic operation> ::=
     { ADD | SUBTRACT | MULTIPLY | DIVIDE | MODULO |
     REM } <result operand> <first source operand>
     <second source operand>
```

Modes :

```
For <monadic operation> is SUCC | PRED :
Order is <first source operand> <second source operand>
     <result operand>
BIT        BIT        BIT
CHAR       CHAR       CHAR
RANGE      RANGE      RANGE

For <monadic operation> is MINUS | ABSOLUTE
Order is <source operand> <result operand>
RANGE      RANGE

For <diadic operation>
Order is <first source operand> <second source operand>
     <result operand>
RANGE      RANGE      RANGE
```

Function :

```
For <monadic operation> is SUCC
<result operand> := successor(
                          value(<source operand>)
```

```
                                    )

    For <monadic operation> is PRED
    <result operand> := predecessor(
                              value(<source operand>)
                                    )

    For <monadic operation> is MINUS
    <result operand> := - value(<source operand>)

    For <monadic operation> is ABSOLUTE
    if value(<source operand>) < 0 then
    <result operand> := - value(<source operand>) else
    <result operand> := value(<source operand>)

    For <diadic operation> is ADD
    <result operand> := value(<first source operand>) +
                              value(<second source operand>)

    For <diadic operation> is SUBTRACT
    <result operand> := value(<first source operand>) -
                              value(<second source operand>)

    For <diadic operation> is MULTIPLY
    <result operand> := value(<first source operand>) *
                              value(<second source operand>)

    For <diadic operation> is DIVIDE
    <result operand> := <first source operand> /
                              <second source operand>

    For <diadic operation> is MODULO
    <result operand> := value(<first source operand>) //
                              value(<second source operand>)

    For <diadic operation> is REM
    <result operand> := remainder (value(<first source operand>) /
                              value(<second source operand>))
```

Pseudo operations :

successor — If I is an integer then successor(I) is I+1. If I+1 is greater than the permitted range the abstract machine generates an error condition.
If C is a character then successor(C) generates the next character in the ordered set of characters. If C is the last character in the set the abstract machine generates an error condition.
If B is a bit then successor(B) generates true if B is false, or an error condition otherwise.

predecessor	If I is an integer then predecessor(I) is I-1. If I-1 is less than the permitted range of I the abstract machine generates an error condition. If C is a character then predecessor(C) generates the previous character in the ordered set of characters. If C is the first character in the set the abstract machine generates an error condition. If B is a bit then predecessor(B) generates false if B is true, or an error condition otherwise.
-	This is arithmetic minus and may be used in modadic or diadic context. In monadic use the effect is identical to the case of diadic zero minus an operand.
+	This is arithmetic addition.
*	This is arithmetic multiplication.
/	This is arithmetic division.
//	This is the modulo operation which is defined as follows. If I and J are integers then I//J delivers a unique integer value K, where 0<=K<J, such that there is an integer value N for which I=N*J+K is true.
remainder	If I and J are integers then remainder(I/J) is the remainder after performing the arithmetic division.

3.5.1.3 Logical Operations

Highly typed languages like Chill and Ada do not permit integers and characters to be treated like bit strings. CORAL 66 does allow this feature. Bit logical operations, eg AND, OR, XOR, are permitted on bits and bit strings only. To cater for this Ivor provides AND, OR, XOR and NOT operations which can be applied to single boolean values or arrays of boolean values, representing bit strings.

Definitions :

```
<logical operation> ::=
     <monadic logical operation> |
     <diadic logical operation>

<monadic logical operation> ::=
     NOT <result operand> <source operand>
```

```
     <diadic logical operation> ::=
          { AND | OR | XOR } <result operand>


          <first source operand> <second source operand>

Modes :
     For <monadic logical operation>
     Order is <source operand> <result operand>
     BIT       BIT
     BIT-ARRAY BIT-ARRAY

     For <diadic logical operation>
     Order is <first source operand> <second source operand>
          <result operand>
     BIT       BIT       BIT
     BIT-ARRAY BIT-ARRAY BIT-ARRAY Both of the source operands and
                                   the result operand must be of
                                   exactly the same mode and
                                   structure.
Function :
     For <monadic logical operation> is NOT
     <result operand> := not(value(<source operand>))

     For <diadic logical operation> is AND
     <result operand> := value(<first source operand>) and
                         value(<second source operand>)

     For <diadic logical operation> is OR
      <result operand> := value(<first source operand>) or
                         value(<second source operand>)

     For <diadic logical operation> is XOR
     <result operand> := value(<first source operand>) exclusive-or
                         value(<second source operand>)

Pseudo operations :
     not            If B is a bit then not(B) obeys the following
                    truth table :

                         B    not(B)
                         0     1
                         1     0

                    If B is an array of bits the operation is
                    performed on all elements of the array
                    treating each element separately.

     and            If B and C are bits then (B and C) obeys the
                    following truth table :

                         B    C    (B and C)
                         0    0       0
                         1    0       0
```

0	1	0
1	1	1

If B and C are arrays of bits the operation is performed on all elements of the array, taking each element in turn, ie
first(B) and first(C) followed by
next(B) and next(C) until
last(B) and last(C).
B and C must have the same number of elements otherwise the abstract machine will generate an error condition.

or — This has identical semantics to the and operation except that the truth table is :

B	C	(B or C)
0	0	0
1	0	1
0	1	1
1	1	1

exclusive-or — This has identical semantics to the and operation except that the truth table is :

B	C	(B exclusive-or C)
0	0	0
1	0	1
0	1	1
1	1	0

3.5.1.4 Numerical Operations

Definitions :

```
<numerical operation> ::=
     <monadic numerical operation> |
     <diadic numerical operation>

<monadic numerical operation> ::=
     { CARD | MAX | MIN | SIZE } <result operand>
     <source operand>

<diadic numerical operation> ::=
     INCLUDE <result operand> <first source operand>
     <second source operand>
```

Modes :

```
For <monadic numerical operation> is CARD | MAX | MIN
Order is <source operand> <result operand>
BIT-ARRAY RANGE

For <monadic numerical operation> is SIZE
Order is <source operand> <result operand>
BIT       RANGE
```

```
     RANGE      RANGE
     CHAR       RANGE
     FLAG       RANGE

     For <diadic numerical operation> :
     Order is <first source operand> <second source operand>
          <result operand>
     BIT-ARRAY BIT-ARRAY BIT  Both  source operands must have the
                              same number of elements.
Function :
     For <monadic numerical operation> is CARD
     <result operand> := card(
                    value(<source operand>)
                                )

     For <monadic numerical operation> is MAX
     <result operand> := maximum(
                              value(<source operand>)
                                    )

     For <monadic numerical operation> is MIN
     <result operand> := minimum(
                              value(<source operand>)
                                    )

     For <monadic numerical operation> is SIZE
     <result operand> := size (
                              value(<source operand>)
                                  )

     For <diadic numerical operation> is INCLUDE
     <result operand> := value(<first source operand>)
          includes value(<second source operand>)
```

Pseudo operations :

card	If B is a bit-array then card(B) is an integer which is the sum of the number of elements of the bit-array which deliver a true value.
maximum	If B is a bit-array then maximum(B) is an integer which is the highest index into the array which delivers a true value.
minimum	If B is a bit-array then minimum(B) is an integer which is the lowest index into the array which delivers a true value.
size	If A is an object of valid mode then size(A) is an integer which gives the number of bits used to hold A.
include	If B and C are bit-arrays of identical

structure then (B includes C) delivers true if, and only if, for each element in C which contains true there is a corresponding true value in the first operand.

3.5.1.5 Complex Mode Accessing Operations

In some of the early programming languages it was not possible to assign many values to array and structure elements without using some kind of looping construction.

It is possible now to use features like :

```
a := [1..10:0,6,b,14..20:0]
```

Which assigns 0 to elements 1 to 10 of array "a", 6 to element 11, the value of "b" to element 12, element 13 is unchanged and elements 14 to 20 are assigned to 0.

Ivor provides operations for creating values like those appearing on the right hand side of the assignment.

In addition, operations are provided for manipulating arrays, eg by slicing a section from an array, or joining 2 arrays, to create a new array. in addition it is possible to create references to array elements and structure fields.

Definitions :

```
<complex mode accessing operation> ::=
     <monadic complex mode accessing operation> |
     <diadic complex mode accessing operation>

<monadic complex mode accessing operation> ::=
     { DEREF | TUPLE } <first operand>
          <second operand>

<diadic complex mode accessing operation> ::=
     { JOIN | SLICE | RANGE | FOR | INDEX | SELECT |
      ATUPLE  } <result operand>
     <first source operand> <second source operand>
```

Modes :

```
For <monadic complex mode accessing operation> is DEREF
Order is <second operand> <first operand>
REF-<basic mode>          <basic mode>
REF-<array mode>          <array mode>
REF-<structure mode>      <structure mode>

For <monadic complex mode accessing operation> is TUPLE
Order is <second operand> <first operand>
<basic mode>   <basic mode>-ARRAY  Length of array depends
                                   upon the number of TUPLE
                                   operations.
```

For <diadic complex mode accessing operation> is JOIN | SLICE
Order is <first source operand> <second source operand>
<result operand>

BIT-ARRAY	BIT-ARRAY	BIT-ARRAY
CHAR-ARRAY	CHAR-ARRAY	CHAR-ARRAY
RANGE-ARRAY	RANGE-ARRAY	RANGE-ARRAY

For <diadic complex mode accessing operation> is RANGE | FOR
Order is <first source operand> <second source operand>
<result operand>

RANGE	RANGE	PAIR

For <diadic complex mode accessing operation> is INDEX
Order is <first source operand> <second source operand>
<result operand>

BIT-ARRAY	RANGE	REF-BIT
CHAR-ARRAY	RANGE	REF-CHAR
RANGE-ARRAY	RANGE	REF-RANGE
FLAG-ARRAY	RANGE	REF-FLAG
LABEL-ARRAY	RANGE	REF-LABEL
PTR-ARRAY	RANGE	REF-PTR
PAIR-ARRAY	RANGE	REF-PAIR
PROC-ARRAY	RANGE	REF-PROC
<array mode>-ARRAY	RANGE	REF-<array mode>

For <diadic complex mode accessing operation> is SELECT
Order is <first source operand> <second source operand>
<result operand>

<structure mode>	<mode construction>	REF-<mode construction>

For <diadic complex mode accessing operation> is ATUPLE
Order is <first source operand> <second source operand>
<result operand>

BIT	RANGE	BIT-ARRAY
CHAR	RANGE	CHAR-ARRAY
RANGE	RANGE	RANGE-ARRAY
FLAG	RANGE	FLAG-ARRAY
LABEL	RANGE	LABEL-ARRAY
PTR	RANGE	PTR-ARRAY
PAIR	RANGE	PAIR-ARRAY
PROC	RANGE	PROC-ARRAY
BIT-ARRAY	PAIR	BIT-ARRAY
CHAR-ARRAY	PAIR	CHAR-ARRAY
RANGE-ARRAY	PAIR	RANGE-ARRAY
FLAG-ARRAY	PAIR	FLAG-ARRAY
LABEL-ARRAY	PAIR	LABEL-ARRAY
PTR-ARRAY	PAIR	PTR-ARRAY
PAIR-ARRAY	PAIR	PAIR-ARRAY
PROC-ARRAY	PAIR	PROC-ARRAY

Function :
For <monadic complex mode accessing operation> is DEREF
<first operand> := value(

```
                    value(<second operand>)
                                )

    For <monadic complex mode accessing operation> is TUPLE
    next (<first operand>) := value(<second operand>)

    For <diadic complex mode accessing operation> is JOIN
    <result operand> := value(<first source operand>) join
                    value(<second source operand>)

    For <diadic complex mode accessing operation> is SLICE
    <result operand> := value(<first source operand>) slice
                    value(<second source operand>)

    For <diadic complex mode accessing operation> is RANGE
    <result operand> := value(<first source operand>)
                  range  value(<second source operand>)

    For <diadic complex mode accessing operation> is FOR
    <result operand> := value(<first source operand>)
                  for  value(<second source operand>)

    For <diadic complex mode accessing operation> is INDEX
    <result  operand> := <first source operand> !
                    value(<second source operand>)

    For <diadic complex mode accessing operation> is SELECT
    <result operand> := <first source operand> .
                    value(<second source operand>)

    For <diadic complex mode accessing operation> is ATUPLE
    <result operand> subarray value(<first source operand>) :=
                    value(<second source operand>)
```

Pseudo operations :

next	If A is an array then next(A) references the highest array element location at which there is no value stored.
join	If A and B are arrays then (A join B) is an new array whose length is the sum of the lengths of A and B.
slice	If A is an array and X is a pair, say (3:4), then (A slice X) will generate an array of length 2 which will be the elements of A at indices 3 and 4.
range	If I and J are integers then (I range J) generates the pair (I:J).
for	If I and J are integers then (I for J) generates the pair (I:I+J).

!	If A is and array reference and X is an integer then (A ! X) is a reference to the Xth element of A.
.	If S is a structure and F is a field of the structure then (S . F) is a reference to the field of the structure.
subarray	If A is an array reference and X is an integer or pair then (A subarray X) references one or more elements of A.

3.5.1.6 Assignment Operation

Definitions :

<assignment operation> ::=
{ ASSIGN | INITIALISE } <sink operand>
<source operand>

Modes :

For <assignment operation>
Order is <source operand> <sink operand>

<mode>	<mode>	Both operands must be of identical mode.

Function :

For <assignment operation> is ASSIGN
<sink operand> := value(<source operand>)

For <assignment operation> is INITIALISE
<sink operand> <- value(<source operand>)

Pseudo operaton :

<-	If A is a reference to an object and B is an object of the same mode as that referenced then (A <- B) causes the value of B to be assigned to the location A once only when the abstract machine first encounters the operation. Thereafter the operation is ignored and causes no action.

3.5.1.7 Result Operation

High level language functions can return a result which may be used in an expression, eg :

```
x := fn()*3+b;
```

It is necessary in Ivor to indicate at the Abstraction introduction (ie abstraction 'fn' in this case) that the abstraction will return a result. A special argument is introduced, the result argument.

In the calling environment the result argument operand is used in

the expression using the procedure result. In the called environment a RESULT operation effectively assigns the value of the result to the result argument operand immediately prior to returning control to the calling environment.

Definitions :

<result operation> ::=
RESULT <procedure result operand>
<result value operand>

Modes :

For <result operation>
Order is <result value operand> <procedure result operand>

<mode>	<mode>	Both operands must be of identical mode.

Function :

For <result operation>
<procedure result operand> := value(<result value operand>)

3.5.1.8 Syncronisation Operation

Definitions :

<syncronisation operation> ::=
SEND <buffer or signal operand>
<receiving process operand> |
RECEIVE <buffer operand>

Modes :

For <syncronisation operation> is SEND
Order is <buffer or signal operand> <receiving process operand>

BUFFER	INSTANCE	
SIGNAL	INSTANCE	Signal is a notion, not a mode, but is included here for completeness.

For <syncronisation operation> is RECEIVE
BUFFER

Function :

For <syncronisation operation> is SEND
send <buffer or signal operand> to <receiving process operand>

For <syncronisation operation> is RECEIVE
<buffer operand> := receive

Pseudo operations :

send — If S is a signal argument and P a process instance value then (send S to P) causes the value of S to be sent to the process instance P.

If B is a buffer value and P a process

instance value then (send B to P) causes the value B to be sent to the process P.

Execution continues with the next instruction immediately.

receive

A concurrently operating process has an input message queue in free space. The receive operation causes the abstract machine to fetch the message at the head of the queue and delete the message from the queue. The message is then available as a value to the abstract machine and may be assigned to a reference of identical mode.

If the queue is empty, the abstract machine will wait until a message is placed in the queue.

3.5.2 Non-Sequential Operations

High level programming languages have procedures, blocks, labels and processes to provide for modular or concurrent program construction, or simply to allow control transfer. The simple high level language statement :

```
proc(p);
```

hides much of the environment creation and parameter passing necessary to implement the feature.

Ivor's design aims at keeping the simplicity of the high level language statement by providing explicit environment activation operations.

Procedures then become simple bracketed operations and a procedure call simply enters at the bracket head.

Definitions :

```
<non-sequential operations> ::=
     <entry operation> |
     <exit operation> |
     <procedure entry operation> |
     <procedure exit operation> |
     <process start operation> |
     <process stop operation> |
     <case selection operation>
```

3.5.2.1 Entry Operation

Definitions :

```
<entry operation> ::=
     <conditional entry operation> |
     <unconditional entry operation>

<conditional entry operation> ::=
     <boolean conditional entry operation> |
     <relational conditional entry operation> |
     <protected entry operation>

<boolean conditional entry operation> ::=
     { ENTERTRUE | ENTERFALSE } <source operand>
     <bracket or label identification key operand>

<relational conditional entry operation> ::=
     { ENTERGT | ENTERGE | ENTERLT | ENTERLE | ENTEREQ | ENTERNE }
     <first source operand> <second source operand>
     <bracket or label identification key operand>

<protected entry operation> ::=
     { ENTERFREE | ENTERBUSY } <source operand>
     <bracket or label identification key operand>
```

```
     <unconditional entry operation> ::=
          ENTER <bracket or label identification key operand>

     <identification key operand> ::=
          <identification key> <operand descriptor>

Modes :
     For <boolean conditional entry operation>
     Order is <source operand>
          <bracket or label identification key operand>
     BIT       LABEL
     BIT       BRACKET

     For <relational conditional entry operation>
     Order is <first source operand> <second source operand>
          <bracket or label identification key operand>
     BIT       BIT       LABEL
     BIT       BIT       BRACKET
     CHAR      CHAR      LABEL
     CHAR      CHAR      BRACKET
     RANGE     RANGE     LABEL
     RANGE     RANGE     BRACKET

     For <protected entry operation>
     Order is <source operand>
          <bracket or label identification key operand>
     FLAG      LABEL
     FLAG      BRACKET

     For <unconditional entry operation>
     LABEL
     BRACKET

Function :
     For <boolean conditional entry operation> is ENTERTRUE
     if value(<source operand>) = TRUE then execute
     <bracket or label identification key operand>

     For <boolean conditional entry operation> is ENTERFALSE
     if value(<source operand>) = FALSE then execute
     <bracket or label identification key operand>

     For <relational conditional entry operation> is ENTERGT
     if value(<first source operand>) >
        value(<second source operand>) then
     execute <bracket or label identification key operand>

     For <relational conditional entry operation> is ENTERGE
     if value(<first source operand>) >=
        value(<second source operand>) then
     execute <bracket or label identification key operand>

     For <relational conditional entry operation> is ENTERLT
     if value(<first source operand>) <
```

```
      value(<second source operand>) then
   execute <bracket or label identification key operand>

   For <relational conditional entry operation> is ENTERLE
   if value(<first source operand>) <=
      value(<second source operand>) then
   execute <bracket or label identification key operand>

   For <relational conditional entry operation> is ENTEREQ
   if value(<first source operand>) =
      value(<second source operand>) then
   execute <bracket or label identification key operand>

   For <relational conditional entry operation> is ENTERNE
   if value(<first source operand>) /=
      value(<second source operand>) then
   execute <bracket or label identification key operand>

   For <protected entry operation> is ENTERFREE
   if value(<source operand>) = FREE then execute
   <bracket or label identification key operand>

   For <protected entry operation> is ENTERBUSY
   if value(<source operand>) = BUSY then execute
   <bracket or label identification key operand>
```

Pseudo operations :

execute	If X is a reference to a location within the Ivor operation stream then (execute X) causes the Ivor instruction interpreter to execute the operation at location X and continue execution from there.

3.5.2.2 Exit Operation

Definitions :

```
   <exit operation> ::=
        <conditional exit operation> |
        <unconditional exit operation>

   <conditional exit operation> ::=
        <boolean conditional exit operation> |
        <relational conditional exit operation> |

   <boolean conditional exit operation> ::=
        { EXITTRUE | EXITFALSE } <source operand>
        <bracket identification key operand>

   <relational conditional exit operation> ::=
        { EXITGT | EXITGE | EXITLT | EXITLE | EXITEQ | EXITNE }
        <first source operand> <second source operand>
        <bracket identification key operand>

   <unconditional exit operation> ::=
```

EXIT <bracket identification key operand>

Modes :

For <boolean conditional exit operation>
Order is <source operand>
<bracket identification key operand>

BIT	BRACKET

For <relational conditional exit operation>
Order is <first source operand> <second source operand>
<bracket identification key operand>

BIT	BIT	BRACKET
CHAR	CHAR	BRACKET
RANGE	RANGE	BRACKET

For <unconditional exit operation>
BRACKET

Function :

For <boolean conditional exit operation> is EXITTRUE
if value(<source operand>) = TRUE then exit
<bracket identification key operand>

For <boolean conditional exit operation> is EXITFALSE
if value(<source operand>) = FALSE then exit
<bracket identification key operand>

For <relational conditional exit operation> is EXITGT
if value(<first source operand>) >
value(<second source operand>) then
exit <bracket identification key operand>

For <relational conditional exit operation> is EXITGE
if value(<first source operand>) >=
value(<second source operand>) then
exit <bracket identification key operand>

For <relational conditional exit operation> is EXITLT
if value(<first source operand>) <
value(<second source operand>) then
exit <bracket identification key operand>

For <relational conditional exit operation> is EXITLE
if value(<first source operand>) <=
value(<second source operand>) then
exit <bracket identification key operand>

For <relational conditional exit operation> is EXITEQ
if value(<first source operand>) =
value(<second source operand>) then
exit <bracket identification key operand>

For <relational conditional exit operation> is EXITNE

```
     if value(<first source operand>) /=
        value(<second source operand>) then
     exit <bracket identification key operand>

     For <protected exit operation> is EXITFREE
     if value(<source operand>) = FREE then exit
     <bracket identification key operand>

     For <protected exit operation> is EXITBUSY
     if value(<source operand>) = BUSY then exit
     <bracket identification key operand>
```

Pseudo operations :

exit	If a section of Ivor operations is bracketed by the bracket X (which has a start of bracket and an end of bracket) then exit causes the instruction interpreter to execute the operations following the closing bracket.

3.5.2.3 Procedure Entry Operation

Definitions :

```
     <procedure entry operation> ::=
          ENTERPROC <procedure operand>
```

Modes :

```
     For <procedure entry operation>
     PROC
```

Function :

```
     For <procedure entry operation>
     execute value(<procedure operand>)
     r:
```

Pseudo Operations

:	See section 3.5.3.2 - labelling

3.5.2.4 Procedure Exit Operation

Definitions :

```
     <procedure exit operation> ::=
          EXITPROC <procedure identification key operand>
```

Modes :

```
     For <procedure exit operation>
     PROC
```

Functions :

```
     For <procedure exit operation>
     deactivate <procedure identification key operand>
     return <procedure identification key operand>
```

Pseudo operations :

deactivate	The deactivate operation does the reverse of the activate operation in that it releases

the free space allocated to the procedure or process.

return — The return operation causes the instruction interpreter to execute the operation (in the operation stream) referenced by the return address planted at the preceding activate instruction.

3.5.2.5 Process Start Operation

Definitions :

<process start operation> ::=
STARTPROCESS <process identification key operand>
<instance identification key operand>

Modes :

For <process start operation>
Order is <process identification key operand>
<instance identification key operand>
PROCESS INSTANCE

Functions :

For <process start operation>
<instance identification key operand> :=
start <process identification key operand>

Pseudo operations :

start — If P is a process and I is an instance object then (I := start P) causes the instruction interpreter to execute P concurrently with the activating process. The activating process continues execution with the operation following the start operation.

In addition a value is returned which is the identity of the instance of the process being executed. This can be assigned to an instance mode object.

3.5.2.6 Process Stop Operation

Definitions :

<process stop operation> ::=
STOPPROCESS

Modes :

None

Functions :

For <process stop operation>
stop

Pseudo operations :

stop	Causes the instruction interpreter to cease execution of the process containing the stop operation. All free space allocated by the activate operation is released and execution ceases. Only the particular process instance is affected by the stop.

3.5.2.7 Case Selection Operation

Ivor caters for source languages having multi-dimensional case statements. Three types of case statements use the same Ivor accessing operations. The type depends upon the mode of the accessing operand.

The first type considers the case to be a 1 to n dimensional array of case labels (with holes, possibly). Control is transferred to the caselabel delivered by indexing the array with the case operands. The case label array is conceptual only as implementation in this manner would not prove feasible.

The remaining types of case statements allow a message to be obtained from one or more message queues, depending upon the queue that contains the message, eg

```
CASE s1,s2,s3 ENDCASE;
```

If only queue "s2" contains a message then the message will be received from "s2". If other queues contain messages then one message will be received though the abstract machine does not define the search order.

Definitions :

```
<case selection operation> ::=
     CASE <case bracket identification key operand>
     <case label array identification key operand>
     <selection> ENDCASE

<selection> ::=
     { RANGE | BUFFER | SIGNAL }
     { <case selector operand> }1:*
```

Modes :

```
For <case selection operation> is CASE <case label
     identification key operand> <case label array
     identification key operand> RANGE { <case
     selector operand> }1:*
Order is <case bracket identification key operand>
     <case label array identification key operand>
     <selection>
CASELABEL      CASELABEL-ARRAY      RANGE

For <case selection operation> is CASE <case label
     identification key operand> <case label array
```

```
          identification key operand> BUFFER { <case
          selector operand> }1:*
     Order is <case bracket identification key operand>
          <case label array identification key operand>
          <selection>
     CASELABEL      CASELABEL-ARRAY     BUFFER

     For <case selection operation> is CASE <case label
          identification key operand> <case label array
          identification key operand> SIGNAL { <case
          selector operand> }1:*
     Order is <case bracket identification key operand>
          <case label array identification key operand>
          <selection>
     CASELABEL      CASELABEL-ARRAY     SIGNAL

Functions :
     For <case selection operation> is CASE <case label
          identification key operand> <case label array
          identification key operand> RANGE  <case
          selector operand>
     execute value(<case label array identification key operand> !
               value(<case selector operand>)

     For <case selection operation> is CASE <case label
          identification key operand> <case label array
          identification key operand> RANGE  <case
          selector 1 operand> <case selector 2 operand>
     execute value( (<case label array identification key operand> !
               value(<case selector 1 operand>) ) !
                    value(case selector 2 operand>) )

     .... and so on.

     For <case selection operation> is CASE <case label
          identification key operand> <case label array
          identification key operand> BUFFER  <case
          selector operand>
     ostack := receive <case selector operand>

     For <case selection operation> is CASE <case label
          identification key operand> <case label array
          identification key operand> BUFFER  <case
          selector 1 operand> <case selector 2 operand>
     if message <case selector 1 operand> then
          ostack := receive <case selector 1 operand>
     orif message <case selector 2 operand> then
          ostack := receive <case selector 2 operand>

     .... and so on

     For <case selection operation> is CASE <case label
          identification key operand> <case label array
          identification key operand> SIGNAL  <case
```

```
          selector operand>
     ostack := receive <case selector operand>

     For <case selection operation> is CASE <case label
          identification key operand> <case label array
          identification key operand> SIGNAL  <case
          selector 1 operand> <case selector 2 operand>
     if message <case selector 1 operand> then
          ostack := receive <case selector 1 operand>
     orif message <case selector 2 operand> then
          ostack := receive <case selector 2 operand>
```

Pseudo operations :

orif — B and Q are boolean and X is an integer. (if B then X:=2 orif Q then X:=3) causes the value of B and the value of Q to be obtained concurrently. This will generate a deterministic result only if B and Q do not change their values during the time taken to evaluate the operation.

The orif condition will be evaluated concurrently with the if condition in addition to any other orif conditionals.

message — If S is a signal or a buffer then (message S) is TRUE if a message is available for receipt by the process from the signal or buffer.

3.5.3 Control Operations

Definitions :

```
     <control operations> ::=
          <bracketing operation> |
          <labelling operation> |
          <environment set up operation> |
          <parameter handling operation> |
          <case association operation> |
          <procedure bracket> |
          <process bracket> |
          <temporary varaible allocation>
```

3.5.3.1 Bracketing Operation

Definitions :

```
     <bracketing operation> ::=
          <open bracket> <operation list> <close bracket>

     <open bracket> ::=
          BRACKET <bracket identification key operand>

     <operation list> ::=
```

```
          { <sequential operation> |
            <non-sequential operation> |
            <control operation> }

     <close bracket> ::=
          ENDBRACKET <bracket identification key operand>

Modes :
     For <open bracket>
     BRACKET

     For <close bracket>
     BRACKET

Functions :
     For <bracketing operation>
     <bracket identification key operand>:
          bracket{ <operation list> }
```

Pseudo operations :

:	The : identifies a point in the operation stream to which control may be transferred. If L is a location within the operation stream then L: identifies the location.
bracket{	The braces {....} identify sections of the operation stream which are bracketed. The identifier preceding the bracket defines the type of the bracket, in this case a simple bracket. Control may be transfered to the start of the bracket by an execute operation and control may be transfered to the end of the bracket by an exit operation. The execute operation may be executed from any point in the program but an exit operation may be executed only from within the bracket which is to be exited.

3.5.3.2 Labelling Operation

```
Definitions :
     <labelling operation> ::=
          LABEL <label identification key operand>

Modes :
     For <labelling operation>
     LABEL

Functions :
     For <labelling operation>
     <label identification key operand> :
```

3.5.3.3 Environment Set Up Operation

Definitions :
<environment set up operation> ::=
ACTIVATE <activation operation>

Modes :
PROC
PROCESS

Functions :
For <environment set up operation>
activate <activation operand>

Pseudo operations :

activate	During the definition interpreter stage the abstract machine assesses the free space requirements for an invocation of a procedure or process. The activate operation allocates the free space and, in the case of a procedure activation, stores within it the address that the instruction interpreter is to execute from (the return address) when it meets a return operation.

3.5.3.4 Parameter Handling Operation

Definitions :
<parameter handling operation> ::=
{ ARGUMENT | OUTARGUMENT } <formal parameter operand>
<actual parameter operand>

Modes :
For <parameter handling operation> with <argument type> of
<formal parameter operand>=IN
Order is <formal parameter operand>
<actual parameter operand>

<basic mode>	<basic mode>	All modes must be identical.
REF-<basic mode>	REF-<basic mode>	All modes must be identical.
<array mode>	<array mode>	All modes must be identical.
<structure mode>	<structure mode>	All modes must be identical.

For <parameter handling operation> with <argument type> of
<formal parameter operand>=INOUT
Order is <formal parameter operand>
<actual parameter operand>

<structure mode>	<basic mode>
<structure mode>	REF-<basic mode>
<strucure mode>	<array mode>
<structure mode>	<structure mode>

Note : If the <actual parameter operand> is of a mode

represented by the identification key "m" then the <formal parameter operand> will be of mode

```
STRUCTURE KEY m KEY fp1 "fp1"
              KEY rm KEYfp2 "fp2"
ENDSTRUCTURE
```

Where :
fp1 is the identification key of the first part of the formal parameter and represents its value.

fp2 is the identification key of the second part of the formal parameter which represents the location of the actual parameter.

rm is the mode of fp2 which is REF m.

For <parameter handling operation> with <argument type> of
<formal parameter operand>=OUT or
<formal parameter operand>=LOC or
<formal parameter operand>=RESULT
Order is <formal parameter operand>
<actual parameter operand>

REF-<basic mode>	<basic mode>	Both <basic mode>s are identical.
REF-REF-<basic mode>	REF-<basic mode>	Both <basic mode>s are identical.
REF-<array mode>	<array mode>	Both <basic mode>s are identical.
REF-<structure mode>	<structure mode>	Both <basic mode>s are identical.

Functions :

For <parameter handling operation> with <argument type> of
<formal parameter operand>=IN
<formal parameter operand> := value(<actual parameter operand>)

For <parameter handling operation> is ARGUMENT
with <argument type> of <formal parameter operand>=INOUT
<formal parameter operand> . fp1 :=
value(<actual parameter operand>)
<formal parameter operand> . :=
<- <actual parameter operand>

For <parameter handling operation> is OUTARGUMENT
with <argument type> of <formal parameter operand>=INOUT
value(<formal parameter operand> . fp2) :=
value(<actual parameter operand>)

For <parameter handling operation> is OUTARGUMENT
with <argument type> of
<formal parameter operand>=OUT or

```
          <formal parameter operand>=LOC or
          <formal parameter operand>=RESULT
     value(<formal parameter operand>) :=
          value(<actual parameter operand>)
```

3.5.3.5 Case Association Operation

In section 3.5.2.7 the case statement was described as a one to "n" dimensional array of caselabels. The case association operation associates a particular caselabel with an element of the n-dimensional array. This could, conceivably, be performed using the assignment statement. In order to permit an efficient implementation, as the n-dimensional array is likely to have many elements without caselabels (ie holes), a special CASE BIND operation is provided.

```
Definitions :
     <case association operation> ::=
          BIND <case label identification key operand>
          <case array identification key operand>
          { <case range pair> }1:* ENDBIND

     <case range pair> ::=
          <lower bound operand> <upper bound operand>

Modes :
     For <case association operation>
     Order is <case label identification key operand>
          <case array identification key operand>
     CASELABEL CASELABEL-ARRAY

     For <case range pair>
     Order is <lower bound operand> <upper bound operand>
     RANGE     RANGE

Functions :
     For <case association operation> is
     BIND <case label identification key operand>
          <case array identification key operand>
          <case range pair>  ENDBIND
     <case array identification key operand> !
     <lower bound operand> to <upper bound operand> :=
          <case label identification key operand>

For <case association operation> is
     BIND <case label identification key operand>
          <case array identification key operand>
          <case range 1 pair>  <case range 2 pair> ENDBIND
     <case array identification key operand> !
     <lower bound 1 operand> !
     <lower bound 2 operand> to <upper bound 2 operand> :=
          <case label identification key operand>
     <case array identification key operand> !
     <lower bound 1 operand> +1 !
```

<lower bound 2 operand> to <upper bound 2 operand> :=
<case label identification key operand>

.... and so on until the first index= <upper bound 1 operand>

... and this is repeated for 3,4,.... n <case range pair> operands.

Pseudo operations :

to	If A is an array then (A ! 1 to 3) represents the 1st, 2nd and 3rd elements of A. The assignment (A! 1 to 3 :=5) assigns the value 5 to the first 3 elements of A.

3.5.2.6 Procedure Bracket

Definitions :

<procedure bracket> ::=
BEGINPROC <procedure identification key operand>
<operation list>
ENDPROC <procedure identification key operand>

Functions :

For <procedure bracket>
<procedure identification key operand>:
procedure{ <operation list> }

Pseudo operations :

procedure{	The braces {....} identify sections of the operation stream which are bracketed. The "procedure" identifier preceding the bracket defines this type of bracket as a procedure bracket. Control may be transfered to the start of the bracket by an execute operation and control may be transfered to the end of the bracket by an exit operation. The execute operation may be executed from any point in the program but an exit operation may be executed only from within the bracket which is to be exited.

3.5.2.7 Process Bracket

Definitions :

<process bracket> ::=
BEGINPROCESS <process identification key operand>
<operation list>
ENDPROCESS <process identification key operand>

Functions :

For <process bracket>
<process identification key operand> :
process{ <operation list> }

Pseudo operations :

process{ — The braces {....} identify sections of the operation stream which are bracketed. The "process" identifier preceding the bracket defines this type of bracket as a process bracket. Control may be transfered to the start of the bracket by a start operation and control may be transfered to the end of the bracket by an exit operation.

3.5.2.8 Temporary Variable Allocation

Definitions :

<temporary variable allocation> ::=
ALLOCATE <temporary key> <mode key> |
FREE <temporary key>

Functions :

For <temporary variable allocation> is ALLOCATE
allocate <temporary key> of <mode key>

For <temporary variable allocation> is FREE
free <temporary key>

Pseudo operations :

allocate .. of — If T is a temporary variable and M is a mode then (allocate T of M) will cause free space to be allocated within the current process or procedure. The free space address allocated will be attributed to T and placed in the data table record of T. Free space allocated in this way is not automatically freed at a deallocate or stop operation and the abstract machine will generate an error condition if temporary storage is not freed before exiting a procedure or process.

free — If T is a temporary variable then (free T) causes the free space allocated to T to be released. The free operation in not permitted on non-temporary variables.

3.6 Constant Stream

Definitions :

```
<constant stream> ::=
     CONSTANTS <constant operations> ENDCONSTANTS

<constant operations. ::=
     { <constant sequential operation> }0:*
```

The constant stream may contain any sequential operation provided:

- The operands are constants in that they can be fully evaluated by the instruction interpreter without it having to access any free space location. An assignment operation will retain the value of the evaluated constant in the data table record of the object being assigned - and not in the object's free space location.

3.7 Conclusions

No other intermediate language considered during this research period had incorporated concurrent processing features, nor did they relate the intermediate code back to the original source from which it was generated. Ivor addressed both of these areas and in addition did not tie the code generator to a particular storage allocation strategy.

The conclusions are :

3.7.1 The concurrent processing facilities would probably require an 'event' type operation where processing is suspended until an event occurs.

3.7.2 More source related information will be required in order to do effective high level debugging at the interpretation stage. Identification of source line and column numbers would be required.

3.7.3 Isolation of storage allocation parameters until code generation stage is viable and sucessfully accomplished in Ivor. The price for this is a more complex code generation phase. This could be replaced by two phases: storage allocation, followed by code generation.

3.8 Example of Use

To illustrate the use of Ivor as an intermediate language the example of figure 2.2.1-1 (reproduced here as figure 3.8-1) will be translated into Ivor.

To simplify the description the following terminology has been adopted :

- Identification key numbers are shown as lower case names.

- Character literals and character strings are shown as their character representation and are enclosed in double quotes, eg "A character string".

- Operand descriptors are omitted and CONSTANT is assumed. Where a VALUE descriptor is appropriate the operand is enclosed in square brackets, eg [KEY k1].

- Operations start on a new line with the operator name occupying the first tabulation position. Operands occupy subsequent tabulation positions and continuation is shown by the first tabulation position being empty.

- Comments start with the characters "--".

In addition declaration of notions, ie labels and brackets have been omitted.

FIGURE 3.8-1

```
002  PROCEDURE fetch element pg;
098  BEGIN
099  INTEGER ARRAY element dl[0:char lit length m];
100  INTEGER char il,kl,hash il,string delimiter il;

103  COMMENT read the element;

105    char il := read a rel char pg;
106  element dl[1] := char il;
107  kl := 2;

109       IF alphabetic m(char il) OR numeric m(char il) THEN
112       BEGIN
114         COMMENT the case of an element being an alpha-numeric
               name or a literal value;
118            FOR char il := read a rel char pg
               WHILE alphabetic m(char il) or numeric m(char il) DO
121            BEGIN
123                 IF irrelevant skipped bg=true m THEN GOTO out1 ll;
125              element dl[kl] := char il;
126              inc m(kl,1);
127            END;
129      out1 ll:
130        END ELSE
132        BEGIN
134          COMMENT case of the element being an operator type (ie
               non alpha-numeric) name or character string;
139          COMMENT check first for character string;
141          string delimiter il := char il;
143             IF string delimiter il = char lit 1 delimiter m OR
144                string delimiter il = char lit 2 delimiter m THEN
146             BEGIN
148               COMMENT this is a character string literal;
151               COMMENT fetch the string and return it ;
153               kl := 1;
155                FOR  char il := ich m
                   WHILE char il <> string delimiter il DO
157                BEGIN
158                  element dl[kl] := char il;
159                  inc m(kl,1);

161                  IF kl>char lit length m THEN
162                  errorhandler(LITERAL(e),
                     "Character%string%literal%is%too%long");

164                  IF char il=new line m THEN
                     inc m(current line ig,1);
166                END;
167                element dl[0]:=kl-1; (Character string length)
168                element type ig := ele char lit m;
169                element ig := LOCATION(element dl[0]);
```

```
170             return m;
171           END;

173         FOR  char il := read a rel char pg
            WHILE non alpha numeric m(char il) DO
175         BEGIN
177           IF irrelavant skipped bg=true m THEN GOTO out211;
179           element dl[kl]:=char il;
180           inc m(kl,1);
181         END;

183       out211:
184       END;

185 back step m;
186 char buff mark ig := char buff mark ig-1;
187 element dl[0] := kl-1;

188 COMMENT at this stage the array 'element dl' is set up such
    that element-0 is the character length of the element and
    element-1 to element-n contain the characters of the
    element;

195 IF numeric m(element dl[1]) THEN
197 BEGIN
199   COMMENT case of a numeric constant being returned ;

201   kl := element dl[0]+1;
202   element ig := 0;

204   FOR kl:=kl-1 WHILE kl>0 DO
206   BEGIN
207     element ig := element ig+power of 10 dg[
        element dl[0]-kl]*element dl[kl];
208   END;

209   element type ig := ele literal m;
210   return m;

211 END ELSE
213 BEGIN
215 COMMENT this is the complex part of the procedure where a
    'name' is to be returned. Firstly it is necessary to search
    the object table to see if the name has been introduced
    already;
221   hash il := 0;
225 search for name ll:

227   IF object string dg[hash il]<> empty m THEN
229   BEGIN
231     COMMENT compare string table with the elements stored
        in 'element dl';
235     FOR kl:=0,kl+1 WHILE kl<= object string dg[hash il] DO
237     BEGIN
```

```
239         IF element dl[kl]<> object string dg[kl+hash il] THEN
241         BEGIN
243           COMMENT the name is not the element so try the next
                name;
245           hash il := hash il+4+object string dg[hash il];
246           repeat m(search for name ll);
247         END;
248       END;

250     COMMENT this is the element name so fetch the object
        table index and return it in 'element ig' whilst
        setting 'element type ig' to 'ele operand m';
256     hash il:=hash il+object string dg[hash il]+1;
257     element ig:=object string dg[hash il]*64*64 +
                    object string dg[hash il+1]*64 +
                    object string dg[hash il+2];
259     element type ig := el operand m;

261     IF in intro bg <> false m AND
           element ig <> rat dot m AND
           element ig <> rat colon m THEN
262     errorhandler pg(LITERAL(e),
           "Duplicate%introduction%of%an%object");
264     END ELSE
266     BEGIN
268       COMMENT this is the case where the name is not known
          and hence the object must be declared;
273       COMMENT check that the static function nesting
          depth is not >0 as declarations are not
          allowed if this is the case;
277       IF func nest depth ig > 0 THEN
278       errorhandler pg(LITERAL(e),
          "Names%cannot%be%introduced%in%a%nested%function%definition");
```

```
281     COMMENT if this is not an introduction (ie USE ...) warn
        the man that we are introducing an object;

285     IF in intro bg=false m THEN
286     errorhandler pg(LITERAL(w),
        "introduction%of%an%object%assumed");

298     obj size m[max obj table mark ig]:=default obj size m;
              (fill in object size)
300     obj str mark m[max obj table mark ig] := hash il;
              (and the place where the name will be kept as
               an index into the object string table.  Use
               the next free object table slot)
306     COMMENT now put the name of the object into the string table;

309     FOR kl:=0,kl+1 WHILE kl<=element dl[0] DO
310     object string dl[hash il+kl]:=element dl[kl];

312     COMMENT and now link back into the object table - into
        3 bytes with the most significant first;
316     inc m(hash il,element dl[0]+1);
317     object string dg[hash il  ]:=byte 2 m(max obj table mark ig);
318     object string dg[hash il+1]:=byte 1 m(max obj table mark ig);
319     object string dg[hash il+2]:=byte 0 m(max obj table mark ig);

321     COMMENT plant code to declare the object;
323     program dg[prog mark ig] := rat declare m;
324     program dg[prog mark ig+1] :=
         byte 2 m(max obj table mark ig);
325     program dg[prog mark ig+2] :=
         byte 1 m(max obj table mark ig);
326     program dg[prog mark ig+3] :=
         byte 0 m(max obj table mark ig);
327     inc m(prog mark ig,4);

329     IF prog mark ig>prog size m THEN errorhandler pg(LITERAL(e),
        "No%program%space%left");

332     COMMENT now set up the return values - firstly 'element ig'
        is set to the object table index of the object and
        'element type ig' is set to 'ele operand m';
337     element ig := max obj table mark ig;
338     element type ig := ele operand m;

340     COMMENT and finally....  tidy up by :
        incrementing 'max obj table mark ig' checking for overflow
        of the object, and object string tables;
344     inc m(max obj table mark ig,2);
346     IF max obj table mark ig>obj table size m OR
347        hash il+2>obj str size m THEN
348     errorhandler pg(LITERAL(e),"Too%many%names%introduced");
```

FIGURE 3.8-2

```
IVOR "mule"
UNIT "fetch element pg"
SEIZED "globals"
   ABSTRACTION SEIZED KEY outer KEY outer OUTERPROCESS
          KEY "outer" outer

--Define the "built in" modes of boolean, byte, integer, character
--and string

       MODE SEIZED KEY outer KEY outer RW BIT KEY boolean
           "boolean" ENDMODE
       MODE SEIZED KEY outer KEY outer RW RANGE 0 255 KEY byte
           "byte" ENDMODE
       MODE SEIZED KEY outer KEY outer RW RANGE -32768 +32767
           KEY integer "integer" ENDMODE
       MODE SEIZED KEY outer KEY outer RW CHAR KEY char "character"
           ENDMODE
       MODE SEIZED KEY outer KEY outer RW ARRAY KEY char 0 255
           KEY string "string" ENDMODE

--Now define the global variables used within the procedure
--firstly "irrelevant skipped bg"
       OBJECT SEIZED KEY outer KEY outer NOTINIT DIRECT KEY boolean
           KEY irrelevantskippedbg "irrelevant skipped bg" ENDOBJECT

--now "current line ig"
       OBJECT SEIZED KEY outer KEY outer LIFEINIT DIRECT KEY integer
           KEY currentlineig "current line ig" ENDOBJECT

-- "element ig"
       OBJECT SEIZED KEY outer KEY outer NOTINIT DIRECT KEY integer
           KEY elementig "element ig" ENDOBJECT

-- "char buff mark ig"
       OBJECT SEIZED KEY outer KEY outer LIFEINIT DIRECT KEY integer
           KEY charbuffmarkig "char buff mark ig" ENDOBJECT
```

```
--  "power of 10 dg" - requires both a mode and object introduction
      MODE SEIZED KEY outer KEY outer RO ARRAY KEY integer 0 10
          KEY mode_powerof10dg "mode_power of 10 dg" ENDMODE
      OBJECT SEIZED KEY outer KEY outer LIFEINIT DIRECT
          KEY mode_powerof10dg KEY charbuffmarkig
          "char buff mark ig" ENDOBJECT

-- "object string dg" - requires both a mode and object introduction
      MODE SEIZED KEY outer KEY outer RW ARRAY KEY char 0 1000
          KEY mode_objectstringdg "mode_object string dg" ENDMODE
      OBJECT SEIZED KEY outer KEY outer NOTINIT DIRECT
          KEY mode_objectstringdg KEY objectstringdg
          "object string dg" ENDOBJECT

-- "in intro bg"
      OBJECT SEIZED KEY outer KEY outer LIFEINIT DIRECT
          KEY boolean KEY inintrobg "in intro bg" ENDOBJECT

-- "obj size m" which is an array of restricted range (to 12 bits).
-- This is not fundamental to the design of the module and was chosen
-- to save space.  Consequently to simplify the description
-- the object has been declared with mode integer.
      MODE SEIZED KEY outer KEY outer RW ARRAY KEY integer 0 1000
          KEY mode_objsizem  "mode_obj size m" ENDMODE
      OBJECT SEIZED KEY outer KEY outer NOTINIT DIRECT
          KEY mode_objsizem KEY objsizem "obj size m" ENDOBJECT

-- "obj str mark m"
      MODE SEIZED KEY outer KEY outer RW ARRAY KEY integer 0 1000
          KEY mode_objstrmarkm "mode_obj str mark m" ENDMODE
      OBJECT SEIZED KEY outer KEY outer NOTINIT DIRECT
          KEY mode_objstrmarkm KEY objstrmarkm "obj str mark m"
          ENDOBJECT

-- "max obj table mark ig"
      OBJECT SEIZED KEY outer KEY outer LIFEINIT DIRECT
          KEY integer KEY maxobjtablemarkig
          "max obj table mark ig" ENDOBJECT
```

```
-- "program dg"
      MODE SEIZED KEY outer KEY outer RW ARRAY KEY byte 0 1000
          KEY mode_programdg "mode_program dg" ENDMODE
      OBJECT SEIZED KEY outer KEY outer NOTINIT DIRECT
          KEY mode_programdg KEY programdg "program dg" ENDOBJECT

-- "prog mark ig"
      OBJECT SEIZED KEY outer KEY outer LIFEINIT DIRECT
          KEY integer KEY progmarkig "prog mark ig" ENDOBJECT

   ENDABSTRACTION
```

```
-- And now define the seized global procedures, and procedure
-- arguments used by the module.

-- Firstly  "read  a rel char pg" which has  one  argument,
-- its  returned result.

   ABSTRACTION SEIZED KEY outer KEY outer PROCEDURE SIMPLE
     NONRECURSIVE NONCRITICAL readarelcharpg "read a rel char pg"

-- And now the procedure arguments, only one the procedure result.
      OBJECT SEIZED KEY readarelcharpg KEY readarelcharpg NOTINIT
          ARGUMENT KEY readarelcharpg RESULT KEY integer
          KEY result_readarelcharpg "result_read a rel char pg"
          ENDOBJECT

-- And now the local variables of the procedure, it has one only
-- "char il".
      OBJECT SEIZED KEY readarelcharpg KEY readarelcharpg
          NOTINIT DIRECT KEY integer KEY charil "char il"
          ENDOBJECT

   ENDABSTRACTION

-- and finally the procedure "error handler pg"
   ABSTRACTION SEIZED KEY outer KEY outer PROCEDURE SIMPLE
     NONRECURSIVE NONCRITICAL errorhandlerpg "error handler pg"

-- And now the procedure arguments, the first and second parameter
      OBJECT SEIZED KEY readarelcharpg KEY readarelcharpg
          NOTINIT ARGUMENT KEY readarelcharpg IN KEY char
          KEY severityq "severity q" ENDOBJECT

      OBJECT SEIZED KEY readarelcharpg KEY readarelcharpg
          NOTINIT ARGUMENT KEY readarelcharpg IN KEY string
          KEY errorstringq "error string q" ENDOBJECT

-- And now the local variables of the procedure, it has one only
-- "kl".
      OBJECT SEIZED KEY readarelcharpg KEY readarelcharpg
          NOTINIT DIRECT KEY integer KEY kl "kl" ENDOBJECT

   ENDABSTRACTION

ENDDEFINITIONS
ENDSEIZED
```

```
-- Now  the definitions of the Ivor unit are introduced in the
-- definition stream.
DEFINITIONS
   ABSTRACTION VISIBLE KEY outer KEY outer PROCEDURE
     SIMPLE NONRECURSIVE NONCRITICAL KEY fetchelementpg
     "fetch element pg"

-- The procedure does not have arguments so introduce ..

-- the objects and modes which are local to the procedure
-- "fetch  element pg"

-- Firstly "element dl"
      MODE VISIBLE KEY fetchelementpg KEY fetchelementpg RW
          ARRAY KEY integer 0 12 KEY mode_elementdl "mode_element dl"
          ENDMODE
      OBJECT VISIBLE KEY fetchelement KEY fetchelement NOTINIT
          DIRECT KEY mode_elementdl KEY elementdl "element dl"
          ENDOBJECT

-- And finally the integers "char il",  "kl", "hash il",
-- "string delimiter il"
      OBJECT VISIBLE KEY fetchelement KEY fetchelement NOTINIT
          DIRECT    KEY integer
                    KEY charil "char il"
                    KEY kl "kl"
                    KEY hashil "hash il"
                    KEY stringdelimiteril "string delimiter il"
          ENDOBJECT

   ENDABSTRACTION
ENDDEFINITIONS

CONSTANTS
-- The constant stream does not contain any operations
ENDCONSTANTS

OPERATIONS
-- The operation stream gives the logic of the program.

-- Line 002 : PROCEDURE fetch element pg;
   BRACKET     KEY fetchelementpg

-- Line 105 : char il := read a rel char pg;
-- The procedure is activated and the result bound to the
-- variable  charil by the ARGUMENT operation.  No assignment
-- is necessary at the procedure return.
   ACTIVATE    KEY readarelcharpg
   ARGUMENT    KEY result_readarelcharpg     KEY charil
   ENTERPROC   KEY readarelcharpg

-- Line 106 : element dl[1] := char il;
-- First evaluate the address of the element onto the address stack
```

```
   INDEX        ADDRESSSTACK KEY refinteger   KEY elementdl
                1 KEY integer
-- Now assign to the address the value of "char il"
   ASSIGN       [ADDRESSSTACK KEY refinteger] [KEY charil]

-- Line 107 : kl := 2;
   ASSIGN       KEY kl                        2 KEY integer

-- Line 109 : IF alphabetic m(char il) OR numeric m(char il ) THEN
-- Note - "alphabetic(x)" is a macro which expands to "x>=LITERAL(A)
-- and x<=LITERAL(z)"
   ENTERLT      [KEY charil]                  "A" KEY char
                KEY intermediate109
   ENTERLE      [KEY charil]                  "Z" KEY char
                KEY true109
   LABEL        KEY intermediate109
-- Note - numeric(x) is a macro which expands to "x>=LITERAL(0) and
-- x<=LITERAL(9)"
   ENTERLT      [KEY charil]                  "0" KEY char
                KEY false109
   ENTERGT      [KEY charil]                  "9" KEY char
                KEY false109
   LABEL        KEY true109

-- Line 118 : FOR char il := read a rel char pg
-- Line 119 :  WHILE alphabetic m(char il) or numeric m(char il) DO
   BRACKET      KEY for118
   ACTIVATE     KEY readarelcharpg
   ARGUMENT     KEY result_readarelcharpg     KEY charil
   ENTERPROC    KEY readarelcharpg
   ENTERLT      [KEY charil]                  "A" KEY char
                KEY intermediate118
   ENTERLE      [KEY charil]                  "Z" KEY char
                KEY true118
   LABEL        KEY intermediate118
   EXITLT       [KEY charil]                  "0" KEY char
                KEY for118
   EXITGT       [KEY charil]                  "9" KEY char
                KEY for118
   LABEL        KEY true118

-- Line 123 : IF irrelevant skipped bg = true m then goto out 1 11;
   ENTERTRUE    [KEY irrelevantskippedbg]     KEY out111

-- Line 125 : element dl[kl] := char il;
   INDEX        ADDRESSSTACK KEY refinteger   KEY elementdl
                [KEY kl]
   ASSIGN       [ADDRESSSTACK KEY refinteger] [KEY charil]

-- Line 126 : inc m(kl,1);
-- Note - "inc m(x,y)" is a macro which expands to "x:=x+y"
   ADD          KEY kl                        [KEY kl]
                1 KEY integer
```

```
-- Line 127 : the end of the FOR loop of line 118
   ENTER        KEY for118
   ENDBRACKET   KEY for118

-- Line 129 : out 1 11:
   LABEL        KEY out111

-- Line 130 : end of the "true" part of the "if....
-- then....else" statement of line 109
   ENTER        KEY fi109
   LABEL        KEY false109

-- Line 141 : string delimiter il := char il;
   ASSIGN       KEY stringdelimiteril          [KEY charil]

-- Line 143 : IF string delimiter il = char lit 1 delimiter m OR
-- Line 144 :    string delimiter il = char lit 2 delimiter m THEN
-- Note - char lit 1 delimiter m is LITERAL(') and
-- char lit 2 delimiter m is LITERAL(")
   ENTEREQ      [KEY stringdelimiteril]        "'" KEY char
                KEY true143
   ENTERNE      [KEY stringdelimiter]          '"' KEY char
                KEY false143
   LABEL        KEY true143

-- Line 153 : kl := 1;
   ASSIGN       KEY kl                         1 KEY integer

-- Line 155 : FOR char il := ich m WHILE char il <>
--            string delimiter il DO
   BRACKET      KEY for155
   ACTIVATE     KEY ichm
   ARGUMENT     KEY result_ichm                KEY charil
   ENTERPROC    KEY ichm
   EXITEQ       [KEY charil]                   [KEY
                                               stringdelimiteril]
                KEY for155

-- Line 158 : element dl[kl] := char il
   INDEX        ADDRESSSTACK KEY refinteger    KEY elementdl
                [KEY kl]
   ASSIGN       [ADDRESSSTACK KEY refinteger] [KEY charil]

-- Line 159 : inc m(kl,1);
   ADD          KEY kl                         [KEY kl]
                1 KEY integer

-- Line 161 : IF kl>char lit length m THEN
-- error handler pg(LITERAL(E),"Character%string%literal
--             %is%too%long");
-- Note - "char lit length m" is a macro representing the
-- integer 63
   ENTERGT      [KEY kl]                       63 KEY integer
                KEY fi161
```

```
   ACTIVATE     KEY errorhandlerpg
   ARGUMENT     KEY severity                    "E" KEY char
   ARGUMENT     KEY errorstring                 "Character%string%
                                                literal%is%too%long"
                                                KEY string
   ENTERPROC    KEY errorhandler
   LABEL        fi161

-- Line 164 : IF char il=new line m THEN inc m(current line ig,1)
-- Note - "new line m" is a macro for the value "-1"
   ENTERNE      [KEY charil]                    -1 KEY integer
                fi164
   ADD          KEY currentlineig               [KEY currentlineig]
                1 KEY integer

-- Line 166 : end of the FOR loop started at line 155
   ENTER        KEY for166
   ENDBRACKET   KEY for166

-- Line 167 : element dl[0] := kl-1;
   SUBTRACT     OPERANDSTACK KEY integer        [KEY kl]
                1 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger     KEY elementdl
                0 KEY integer
   ASSIGN       [ADDRESSSTACK KEY refinteger]   [OPERANDSTACK
                                                KEY integer]

-- Line 168 : element type ig := ele char lit m;
-- Note - "ele char lit m" is a macro for the value "2"
   ASSIGN       KEY elementtypeig               2 KEY integer
```

```
-- Line 169 : element ig := LOCATION(element dl[0]);
   INDEX       ADDRESSSTACK KEY refinteger   KEY elementdl
               0 KEY integer
   ASSIGN      KEY elementig                 ADDRESSSTACK
                                             KEY refinteger

-- Line 170 : return m;
-- Note - CORAL 66 does not have a procedure return statement.
-- The macro "return m" has been used consistently to mean
-- return from the procedure and will be interpreted as meaning
-- this.
   EXITPROC    KEY fetchelementpg

-- Line 171 : end of the IF statement of line 143
   LABEL       fi143

-- Line 173 : FOR char il := read a rel char pg WHILE
--            non alpha numeric m(char il) DO
-- Note - The macro "non alpha numeric m(x)" is translated into
-- "x<LITERAL(A) OR x>LITERAL(Z) AND x<LITERAL(0) OR x>LITERAL(9)"
   BRACKET     KEY for173
   ACTIVATE    KEY readarelcharpg
   ARGUMENT    KEY result_readarelcharpg     KEY charil
   ENTERPROC   KEY readarelcharpg
   ENTERLT     [KEY charil]                  "A" KEY char
               KEY intermediate173
   EXITLE      [KEY charil]                  "Z" KEY char
               KEY for173
   LABEL       KEY intermediate173
   ENTERLT     [KEY charil]                  "0" KEY char
               KEY true173
   EXITGT      [KEY charil]                  "9" KEY char
               KEY for173
   LABEL       KEY true173

-- Line 177 : IF irrelevant skipped bg=true m THEN
--            GOTO out 2 11;
   ENTERTRUE   [KEY irrelevantskippedbg]     KEY out211

-- Line 179 : element dl [kl] := char il;
   INDEX       ADDRESSSTACK KEY refinteger   KEY elementdl
               [KEY kl]
   ASSIGN      [ADDRESSSTACK KEY refinteger] [KEY charil]

-- Line 180 : inc m(kl,1);
   ADD         KEY kl                        [KEY kl]
               1 KEY integer
```

```
-- Line 181 : end of FOR loop of line 173
   ENTER        KEY for173
   ENDBRACKET   KEY for173

-- Line 183 : out 2 11:
   LABEL        KEY out211

-- Line 184 : end of false part of the IF statement of line 109
   LABEL        KEY fi109

-- Line 185 : back space m;
   ACTIVATE     KEY backspacem
   ENTERPROC    KEY backspacem

-- Line 186 : char buff mark ig := char buff mark ig -1;
   SUBTRACT     KEY charbuffmarkig              [KEY charbuffmarkig]
                1 KEY integer

-- Line 187 : element dl[0] := kl-1;
   SUBTRACT     OPERANDSTACK KEY integer        [KEY kl]
                1 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger     KEY elementdl
                0 KEY integer
   ASSIGN       [ADDRESSSTACK KEY refinteger]   [OPERANDSTACK
                                                KEY integer]

-- Line 195 : IF numeric m(element dl[1]) THEN
   INDEX        ADDRESSSTACK KEY refinteger     KEY elementdl
                1 KEY integer
   ALLOCATE     KEY temporary195 KEY char
   DEREF        KEY temporary195                [ADDRESSSTACK
                                                KEY refinteger]
   ENTERGE      KEY temporary195                "0" KEY char
                KEY true195
   ENTERGT      KEY temporary195                "9" KEY char
                KEY false195
   LABEL        KEY true195

-- Line 201 : kl := element dl[0]+1;
   INDEX        ADDRESSSTACK KEY refinteger     KEY elementdl
                0 KEY integer
   DEREF        OPERANDSTACK KEY integer        [ADDRESSSTACK
                                                KEY refinteger]
   ADD          KEY kl                          [OPERANDSTACK
                                                KEY integer]
                1 KEY integer

-- Line 202 : element ig := 0;
   ASSIGN       KEY elementig                   0 KEY integer

-- Line 204 : FOR kl := kl-1 WHILE kl>0 DO
   BRACKET      KEY for204
   SUBTRACT     KEY kl                          [KEY kl]
                1 KEY integer
```

```
   EXITLE       [KEY kl]                          0 KEY integer
                KEY for204

-- Line 207 : element ig := element ig+power of 10 dg
--              [element dl[0]-kl]*element dl[kl]
   INDEX        ADDRESSSTACK KEY refinteger       KEY elementdl
                0 KEY integer
   DEREF        OPERANDSTACK KEY integer          [ADDRESSSTACK
                                                  KEY refinteger]
   SUBTRACT     OPERANDSTACK KEY integer          [OPERANDSTACK
                                                  KEY integer]
                [KEY kl]
   INDEX        ADDRESSSTACK KEY refinteger       KEY elementdl
                [KEY kl]
   DEREF        OPERANDSTACK KEY integer          [ADDRESSSTACK
                                                  KEY refinteger]
   MULTIPLY     OPERANDSTACK KEY integer          [OPERANDSTACK
                                                  KEY integer]
                [OPERANDSTACK KEY integer]
   ADD          KEY elementig                     [KEY elementig]
                [OPERANDSTACK KEY integer]

-- Line 208 : end of FOR loop of line 204
   ENTER        for204
   ENDBRACKET   for204

-- Line 211 : end of true part of IF statement of line 195
   ENTER        fi195
   LABEL        false195

-- Line 221 : hash il := 0;
   ASSIGN       KEY hashil                        0 KEY integer

-- Line 225 : seach for name ll:
   LABEL        KEY searchfornamell

-- Line 227 : IF object string dg[hash il]<> empty m THEN
-- Note - "empty m" is a macro representing "-1"
   INDEX        ADDRESSSTACK KEY refinteger       KEY objectstringdg
                [KEY hashil]
   DEREF        OPERANDSTACK KEY integer          [ADDRESSSTACK
                                                  KEY refinteger]
   ENTEREQ      [OPERANDSTACK KEY integer]        -1 KEY integer
                KEY fi227

-- Line 235 : FOR kl := 0,kl+1 WHILE kl <=
--             object string dg[hash il] DO
   BRACKET      for235
   ASSIGN       KEY kl                            0 KEY integer
   ENTER        KEY forstart235
   LABEL        KEY forelement2_235
   ADD          KEY kl                            [KEY kl]
                1 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger       KEY objectstringdg
```

```
                [KEY hashil]
   DEREF        OPERANDSTACK KEY integer        [ADDRESSSTACK
                                                KEY refinteger]
   EXITGT       [KEY kl]                        [OPERANDSTACK
                                                KEY integer]
                KEY for235
   LABEL        KEY forstart235

-- Line 239 : IF element dl[kl]<>object string dg[kl+hash il]
--              THEN
   INDEX        ADDRESSSTACK KEY refinteger     KEY elementdl
                [KEY kl]
   DEREF        OPERANDSTACK KEY integer        [ADDRESSSTACK
                                                KEY refinteger]
   ADD          OPERANDSTACK KEY integer        [KEY kl]
                [KEY hashil]
   INDEX        ADDRESSSTACK KEY refinteger     KEY objectstringdg
                [OPERANDSTACK KEY integer]
   DEREF        OPERANDSTACK KEY integer        [ADDRESSSTACK
                                                KEY refinteger]
   ENTEREQ      [OPERANDSTACK KEY integer]      [OPERANDSTACK
                                                KEY integer]
                fi239

-- Line 245 : hash il := hash il+4+object string dg[hash il];
   ADD          OPERANDSTACK KEY integer        [KEY hashil]
                4 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger     KEY objectstringdg
                [KEY hashil]
   DEREF        OPERANDSTACK KEY integer        [ADDRESSSTACK
                                                KEY refinteger]
   ADD          KEY hashil                      [OPERANDSTACK
                                                KEY integer]
                [OPERANDSTACK KEY integer]

-- Line 246 : repeat m(search for name 11);
-- Note - "repeat m(x)" is a macro which expands to "GOTO m"
   ENTER        KEY searchfornamell

-- Line 247 : end of IF statement of line 239
   LABEL        KEY fi239

-- Line 248 : end of FOR statement of line 235
   ENTER        KEY forelement2_235
   ENDBRACKET   KEY for235

-- Line 256 : hash il := hash il + object string dg[hash il]+1;
   INDEX        ADDRESSSTACK KEY refinteger     KEY objectstringdg
                [KEY hashil]
   DEREF        OPERANDSTACK KEY integer        [ADDRESSSTACK
                                                KEY refinteger]
   ADD          OPERANDSTACK KEY integer        [OPERANDSTACK
                                                KEY integer]
                1 KEY integer
```

```
   ADD          KEY hashil                          [KEY hashil]
                [OPERANDSTACK KEY integer]

-- Line 257 : element ig := object string dg[hash il]*64*64+
--                          object string dg[hash il+1]*64 +
--                          object string dg[hash il+2];
   INDEX        ADDRESSSTACK KEY refinteger    KEY objectstringdg
                [KEY hashil]
   DEREF        OPERANDSTACK KEY integer       [ADDRESSSTACK
                                               KEY refinteger]
   MULTIPLY     OPERANDSTACK KEY integer       [OPERANDSTACK
                                               KEY integer]
                4096 KEY integer
   ADD          OPERANDSTACK KEY integer       [KEY hashil]
                1 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger    KEY objectstringdg
                [OPERANDSTACK KEY integer]
   DEREF        OPERANDSTACK KEY integer       [ADDRESSSTACK
                                               KEY refinteger]
   MULTIPLY     OPERANDSTACK KEY integer       [OPERANDSTACK
                                               KEY integer]
                64 KEY integer
   ADD          OPERANDSTACK KEY integer       [OPERANDSTACK
                                               KEY integer]
                [OPERANDSTACK KEY integer]
   ADD          OPERANDSTACK KEY integer       [KEY hashil]
                2 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger    KEY objectstringdg
                [OPERANDSTACK KEY integer]
   DEREF        OPERANDSTACK KEY integer       [ADDRESSSTACK
                                               KEY refinteger]
   ADD          KEY elementig                  [OPERANDSTACK
                                               KEY integer]
                [OPERANDSTACK KEY integer]
-- Line 259 : element type ig := ele operand m;
-- Note - "ele operand m" is a macro for the value "1"
   ASSIGN       KEY elementtypeig              1 KEY integer

-- Line 261 : IF in intro bg <> false m AND
--                element ig  <> rat dot m AND
--                element ig  <> rat colon m THEN
-- Note - "rat dot m" is a macro for the value "2", and
--        "rat colon m" is a macro for the value "0"
   ENTERTRUE    [KEY inintrobg]                KEY fi261
   ENTEREQ      [KEY elementig]                2 KEY integer
                KEY fi261
   ENTEREQ      [KEY elementig]                0 KEY integer
                KEY fi261

-- Line 262 : errorhandler pg(LITERAL(E),"Duplicate%
--                 introduction%of%an%object");
--              ... and the end of the IF statement of line 261
   ACTIVATE     KEY errorhandlerpg
   ARGUMENT     KEY severity                   "E" KEY char
```

```
   ARGUMENT     KEY errorstring                  "Duplicate%introduction
                                                 %of%an%object"
                                                 KEY string
   ENTERPROC    KEY errorhandler
   LABEL        KEY fi261

-- Line 264 : end of the true part of the IF statement of line 227
   ENTER        KEY fi227
   LABEL        KEY false227

-- Line 277 : IF func nest depth ig > 0 THEN
--                 errorhandler pg(LITERAL(E),"Names%cannot%be%
--                 introduced%in%a%nested%function%definition");
   ENTERLE      [KEY funcnestdepthig]            0 KEY integer
                KEY fi277
   ACTIVATE     KEY errorhandlerpg
   ARGUMENT     KEY severity                     "E" KEY char
   ARGUMENT     KEY errorstring                  "Names%cannot%be%
                introduced%in%a%nested%function%definition"
                                                 KEY string
   ENTERPROC    KEY errorhandler
   LABEL        KEY fi277
```

```
-- Line 285 : IF in intro bg=false m THEN
--   errorhandler pg(LITERAL(W),"Introduction%of%an%
--   object%assumed");
   ENTERFALSE  [KEY inintrobg]                  KEY fi285
   ACTIVATE    KEY errorhandlerpg
   ARGUMENT    KEY severity                     "E" KEY char
   ARGUMENT    KEY errorstring                  "Introduction%of%an%
                                                object%assumed"
                                                KEY string
   ENTERPROC   KEY errorhandler
   LABEL       KEY fi285

-- Line 298 : obj size m[max obj table mark ig] :=
-- default obj size m;
-- Note - "default obj size m" is a macro for the value "4"
   INDEX       ADDRESSSTACK KEY refinteger   KEY objsizem
               [KEY maxobjtablemarkig]
   ASSIGN      [ADDRESSSTACK KEY refinteger] 4 KEY integer

-- Line 300 : obj str mark m[max obj table mark i] := hash il;
   INDEX       ADDRESSSTACK KEY refinteger   KEY objstrmarkm
               [KEY maxobjtablemarkig]
   ASSIGN      [ADDRESSSTACK KEY refinteger] KEY hashil

-- Line 309 : FOR kl:=0,kl+1 WHILE kl<=element dl[0] DO
--              object string dl[hash il] := element dl[kl];
   BRACKET     KEY for309
   ASSIGN      KEY kl                           0 KEY integer
   ENTER       KEY forstart309
   LABEL       KEY forelement2_309
   ADD         KEY kl                           [KEY kl]
               1 KEY integer
   INDEX       ADDRESSSTACK KEY refinteger   KEY elementdl
               0 KEY integer
   DEREF       OPERANDSTACK KEY integer      [ADDRESSSTACK
                                             KEY refinteger]
   EXITGT      [KEY kl]                      [OPERANDSTACK
                                             KEY integer]
               KEY for309
   LABEL       KEY forstart309
   INDEX       ADDRESSSTACK KEY refinteger   KEY objectstringdl
               [KEY hashil]
   INDEX       ADDRESSSTACK KEY refinteger   KEY elementdl
               [KEY kl]
   DEREF       OPERANDSTACK KEY integer      [ADDRESSSTACK
                                             KEY refinteger]
   ASSIGN      [ADDRESSSTACK KEY refinteger] [OPERANDSTACK
                                             KEY integer]
   ENTER       KEY forelement2_309
   ENDBRACKET  KEY for309

-- Line 316 : inc m(hash il,element dl[0]+1);
   INDEX       ADDRESSSTACK KEY refinteger   KEY elementdl
               0 KEY integer
```

```
   DEREF        OPERANDSTACK KEY integer       [ADDRESSSTACK
                                               KEY refinteger]
   ADD          OPERANDSTACK KEY integer       [OPERANDSTACK
                                               KEY integer]
                1 KEY integer
   ADD          KEY hashil                     [KEY hashil]
                [OPERANDSTACK KEY integer]

-- Line 317 : object string dg[hash il] :=
--            byte 2 m(max obj table mark ig);
-- Note "byte 2 m(x)" is a macro which delivers as an integer bits
-- 12 to 17 (if the LSB is bit 0) of "x"
   INDEX        ADDRESSSTACK KEY refinteger    KEY objectstringdg
                [KEY hashil]
   RANGE        OPERANDSTACK KEY pair          12 KEY integer
                17 KEY integer
   CONVERT      OPERANDSTACK KEY intbitstring KEY maxobjstringdg
   SLICE        OPERANDSTACK KEY intbitstring [OPERANDSTACK
                                               KEY intbitstring]
                [OPERANDSTACK KEY pair]
   CONVERT      OPERANDSTACK KEY integer       [OPERANDSTACK
                                               KEY intbitstring]
   ASSIGN       [ADDRESSTACK KEY refinteger]   [OPERANDSTACK
                                               KEY integer]

-- Line 318 : object string dg[hash il+1] :=
--            byte 1 m(max obj table mark ig);
-- Note "byte 1 m(x)" is a macro which delivers as an integer bits
-- 6 to 11 (if the LSB is bit 0) of "x"
   ADD          OPERANDSTACK KEY integer       [KEY hashil]
                1 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger    KEY objectstringdg
                [OPERANDSTACK KEY integer]
   RANGE        OPERANDSTACK KEY pair          6 KEY integer
                11 KEY integer
   CONVERT      OPERANDSTACK KEY intbitstring KEY maxobjstringdg
   SLICE        OPERANDSTACK KEY intbitstring [OPERANDSTACK
                                               KEY intbitstring]
                [OPERANDSTACK KEY pair]
   CONVERT      OPERANDSTACK KEY integer       [OPERANDSTACK
                                               KEY intbitstring]
   ASSIGN       [ADDRESSTACK KEY refinteger]   [OPERANDSTACK
                                               KEY integer]

-- Line 319 : object string dg[hash il+2] :=
--            byte 0 m(max obj table mark ig);
-- Note "byte 0 m(x)" is a macro which delivers as an integer bits
-- 0 to 5 (if the LSB is bit 0) of "x"
   ADD          OPERANDSTACK KEY integer       [KEY hashil]
                2 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger    KEY objectstringdg
                [OPERANDSTACK KEY integer]
   RANGE        OPERANDSTACK KEY pair          0 KEY integer
                5 KEY integer
```

```
   CONVERT     OPERANDSTACK KEY intbitstring KEY maxobjstringdg
   SLICE       OPERANDSTACK KEY intbitstring [OPERANDSTACK
                                             KEY intbitstring]
               [OPERANDSTACK KEY pair]
   CONVERT     OPERANDSTACK KEY integer      [OPERANDSTACK
                                             KEY intbitstring]
   ASSIGN      [ADDRESSTACK KEY refinteger]  [OPERANDSTACK
                                             KEY integer]

-- Line 323 : program dg[prog mark ig] := rat declare m;
-- Note - "rat declare m" is a macro for the value "5"
   INDEX       ADDRESSSTACK KEY refinteger   KEY programdg
               [KEY progmarkig]
   ASSIGN      [ADDRESSSTACK KEY refinteger] 5 KEY integer

-- Line 324 : program dg[progmark ig +1] :=
--             byte 2 m(max obj table mark ig);
   ADD          OPERANDSTACK KEY integer     [KEY progmarkig]
                1 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger  KEY programdg
                [OPERANDSTACK KEY integer]
   RANGE        OPERANDSTACK KEY pair        12 KEY integer
                17 KEY integer
   CONVERT      OPERANDSTACK KEY intbitstring KEY maxobjstringdg
   SLICE        OPERANDSTACK KEY intbitstring [OPERANDSTACK
                                              KEY intbitstring]
                [OPERANDSTACK KEY pair]
   CONVERT      OPERANDSTACK KEY integer      [OPERANDSTACK
                                              KEY intbitstring]
   ASSIGN       [ADDRESSTACK KEY refinteger]  [OPERANDSTACK
                                              KEY integer]

-- Line 325 : program dg[progmark ig +2] :=
--             byte 1 m(max obj table mark ig);
   ADD          OPERANDSTACK KEY integer     [KEY progmarkig]
                2 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger  KEY programdg
                [OPERANDSTACK KEY integer]
   RANGE        OPERANDSTACK KEY pair        6 KEY integer
                11 KEY integer
   CONVERT      OPERANDSTACK KEY intbitstring KEY maxobjstringdg
   SLICE        OPERANDSTACK KEY intbitstring [OPERANDSTACK
                                              KEY intbitstring]
                [OPERANDSTACK KEY pair]
   CONVERT      OPERANDSTACK KEY integer      [OPERANDSTACK
                                              KEY intbitstring]
   ASSIGN       [ADDRESSTACK KEY refinteger]  [OPERANDSTACK
                                              KEY integer]

-- Line 326 : program dg[progmark ig +3] :=
--             byte 0 m(max obj table mark ig);
   ADD          OPERANDSTACK KEY integer     [KEY progmarkig]
                3 KEY integer
   INDEX        ADDRESSSTACK KEY refinteger  KEY programdg
```

```
                [OPERANDSTACK KEY integer]
   RANGE        OPERANDSTACK KEY pair          0 KEY integer
                5 KEY integer
   CONVERT      OPERANDSTACK KEY intbitstring KEY maxobjstringdg
   SLICE        OPERANDSTACK KEY intbitstring [OPERANDSTACK
                                              KEY intbitstring]
                [OPERANDSTACK KEY pair]
   CONVERT      OPERANDSTACK KEY integer       [OPERANDSTACK
                                              KEY intbitstring]
   ASSIGN       [ADDRESSTACK KEY refinteger]  [OPERANDSTACK
                                              KEY integer]

-- Line 329 : IF prog mark ig > prog size m THEN
--   errorhandler pg(LITERAL(E),"No%program%space%left");
-- Note "prog size m" is a macro for the value "10000"
   ENTERLE      [KEY progmarkig]              10000 KEY integer
                KEY fi329
   ACTIVATE     KEY errorhandlerpg
   ARGUMENT     KEY severity                  "E" KEY char
   ARGUMENT     KEY errorstring               "No%program%
                                              space%left"
                                              KEY string
   ENTERPROC    KEY errorhandler
   LABEL        KEY fi329

-- Line 337 : element ig := max obj table mark ig;
   ASSIGN       KEY elementig                 [KEY
                                              maxobjtablemarkig]

-- Line 338 : element type ig := ele operand m;
   ASSIGN       KEY elementtypeig             1 KEY integer
```

```
-- Line 344 : inc m(max obj table mark ig,2);
   ADD          KEY maxobjtablemarkig          [KEY
                                               maxobjtablemarkig]
                2 KEY integer

-- Line 346 : IF max obj table mark ig > obj table size m OR
--                hash il +2 > obj str size m THEN
--      errorhandler pg(LITERAL(E),"Too%many%names%introduced");
-- Note - "obj table size m" is a macro for the value "1000" and
--          "obj str size m" is a macro for the value "10000"
   ENTERGT      [KEY maxobjtablemarkig]        1000 KEY integer
                KEY true346
   ADD          OPERANDSTACK KEY integer       [KEY hashil]
                2 KEY integer
   ENTERLE      [OPERANDSTACK KEY integer]     10000 KEY integer
                KEY fi346
   LABEL        true346
   ACTIVATE     KEY errorhandlerpg
   ARGUMENT     KEY severity                   "E" KEY char
   ARGUMENT     KEY errorstring                "Too%many%names%
                                               introduced"
                                               KEY string
   ENTERPROC    KEY errorhandler
   LABEL        KEY fi346

-- Line 349 : end of false part of IF statement of line 227
   LABEL        KEY fi227

-- Line 350 : end of false part of IF statement of line 195
   LABEL        KEY fi195

-- Line 353 : end of procedure, and end of module
   EXITPROC     KEY fetchelementpg
   ENDBRACKET   KEY fetchelementpg

ENDOPERATIONS

ENDUNIT
```

4 MULE

This section of the Thesis describes a function based programming language called Mule. Mule was designed and implemented as part of the research for this degree. The aims were :

- To provide a programming environment rather than a language.

- To design the environment to be portable.

- To provide an environment which can evolve.

- To implement a pilot Mule environment in order to test out the idea.

In many of the program portability methods researched portability was achieved by reprocessing the program being ported using either a compiler, macroprocessor, interpreter, etc in the new environment. This gave the user of the program a new environment to work in in addition to the program operating in the new environment. For example, if a spread-sheet program is transported from an IBM machine environment to an ICL machine environment then only the use of the spread-sheet (assuming a successful port) remains the same. The user needs to learn (possibly) new logging on sequences, new ways of invoking programs, and possibly new ways of invoking simple utilities like editing.

A way of removing this problem is to provide a user with an environment in which he works. The environment would provide all of the facilities that he requires and he would never need to venture outside of the environment, nor need to access any of the native facilities of the underlying host machine.

So, if the user wishes to use his facility set on new machine it would be necessary to move the whole environment to that new machine.

This section of the research was aimed at investigating the needs of such an environment and looking at ways of making it easy to port.

In order to meet the aims it was decided to define a kernel of facilities to form the foundation of the mule environment. The kernel will provide 1st-level facilities. In order to provide a greater number of facilities, say 2nd-level facilities, Mule could be enhanced using 1st-level facilities alone. So the facilities available to a user would be both 1st and 2nd-level facilities. In order to provide a greater range of facilities still the Mule environment could be enhanced to provide, say 3rd-level facilities, using the facilities of the 1st-level and 2nd-level ... and so on.

The success, or otherwise, of this approach depends upon the design of the kernel which should provide sufficient facilities to allow the upper levels (ie 2nd, 3rd ... level facilities) to be implemented but should be sufficiently small so that it can be transported with relative ease.

The real success of such an environment, however, is in the ability to produce applications on top of the basic environment. The Mule language designed is probably rich enough for this but this was not tested to any depth, except for the production of language additions.

A Mule environment was designed and implemented and this is described in the following sections, as follows :

- The Mule language
- The Pilot Implementation
- Test programs
- Conclusions

The major conclusions are :

- That it is easy to add new functions to Mule and that these functions are identical to built-in or kernel functions, as viewed by the Mule progammer.
- The concept of mode or type would be a useful addition and could be added using kernel facilities alone.
- The kernel is small, but sufficient.
- The lexical analyser should be produced in Mule to obtain maximum transportability.

4.1 The Mule Language

The idea was to develop a function based language which had very few basic concepts and a very simple underlying abstract machine.

The function was to be the way that facilities were provided. Some base level functions were to form the Mule kernel and were to be used in order to create higher level functions which could, in the end provide applications, eg editing, data handling.. etc.

Little consideration was to be given to the syntax of the language as it was the idea of a hierarchy of functions that was to be addressed, rather than a syntactically elegant language.

What follows is a description of the design work performed in order to meet these aims, and details about the Mule pilot implementation.

The mule language assumes a very simple underlying machine with 4 data areas : data memory, program memory, a stack and an object table.

Data memory is the main memory for holding the values of introduced objects. Program memory holds the Mule function definitions. Data and Program memory are totally separate ensuring that data can never be treated as program and inadvertantly executed.

Data memory comprises cells, each of identical (but unspecified) bit size and with addresses starting at cell 0 and increasing by 1.

Program memory comprises cells, again with each having identical (but unspecified) bit size and with addresses starting at cell 0 and increasing by 1.

The stack is used for holding values during expression evaluation.

The stack is a first-in last-off organised data structure which can be used to hold values. The Mule kernel provides the programmer with facilities for adding objects to the stack and for removing them.

The object table holds information about objects which are introduced, or used within a Mule program. Objects will be described later but for now consider them to be the names of variables and subroutines. Access to the object table is obtained by using the object's name as a key. The information held in the object table is the start address of the object (ie memory cell address) and the size of the object, in memory cells.

The Mule language will be described using the modified BNF of

section 2. To simplify the syntax description space-character and new line separators are not shown in the syntax description. Both spaces and new lines are significant and multiple spaces, or new lines, are exactly equivalent. In general spaces and new lines are required wherever it would be ambiguous not to so do.

For example: USE A B. requires the space between the USE and the name A and between the name A and name B.
USEA B. is ambiguous, and ...
USE AB. intoduces a single name, ie AB.

```
<mule language> ::=
     {<object introduction> |
      <stacking operation> |
      <function call operation> |
      <function stacking operation> |
      <infix stacking operation> |
      <destacking operation> }0:*
```

The operations will be described in more detail in subsequent section. They allow :

- Objects to be introduced and then used subsequently in other operations.

- The results of, for example arithmetic or logical operations, to be evaluated and placed on the stack.

- A value to be placed on the stack.

- A function to be defined and placed upon the stack.

- A function, which has been placed upon the stack, to be executed.

- A value to be removed from the stack and placed in data memory.

Objects may be introduced, though they need not be. They are given names in a similar manner to other programming languages, eg a vaid name might be:

CASHFLOW or PRINCIPAL

A number of data memory cells are allocated to the object and a pointer to the first memory cell is placed in the data table record for the introduced object.

The number of data memory cells is dependent upon the manner in which the object is introduced. For example if an object is introduced by :

USE cashflow:25.

Then 25 data memory cells will be allocated to the object. The introduction is executed every time it is met so that memory is allocated and released as the program executes. So, an introduction of the form :

```
USE principal:scale.
```

Will allocate a variable number of data memory cells, depending upon the value of the object 'scale'. There are defaults too. If an object is introduced without the size being specified (or if the object is not formally introduced at all) a default size of four data memory cells is used.

Object names are not restricted to the more common kind of names, like those shown above. Special names can be introduced too, eg:

```
USE ++ , $$ .
```

These names can be used in exactly the same manner as "normal" names but they are provided in order to allow the Mule environment to be enhanced by the addition of new operators, by using the introduced names in order to invoke Mule functions. Similarly, special names may have a size attribute specified.

Fundamental to the operation of Mule is the stack. Functions do not have parameters (as such) and (strictly) there are no expressions. Mule operations take items from the stack and return items to the stack. An item may be an object's value, an object's address or a constant value. The size of an object is an integral part of its identity and when an object is placed on the stack the top stack item assumes the size attribute of the item being placed on the stack. Again, there are defaults. For example an integer constant has a size of four memory cells, as does an object address.

Functions can be defined in Mule by the use of the FUNCTION and END brackets. The effect of the definition is to, effectively, place the function definition on the stack. For example,

```
FUNCTION 123.72 END
```

is a function definition which, WHEN EXECUTED, will place 123.72 on the stack. A more useful example is :

```
FUNCTION
  USE localstring:stringsize
  -> @localstring
  @localstring; outchar()
END
```

This function, when executed, takes a value from the stack and assigns it to the data memory locations of the object "localstring". It then prints the string via the built in function "outchar"

The function defined has no name. It is merely placed on the stack. Functions can be executed from the stack by using the function call operation, eg

```
FUNCTION 1234.72 END ()
```

will cause a function to be placed on the stack which will be immediately executed to cause the constant value, 1234.72, to be placed on the stack.

It is more normal to assign a function to an introduced object. The object must have a size attribute of the default, ie 4 memory cells. In the previous case a new function, "outstring", can be created by assigning the function, as follows :

```
USE outstring.
FUNCTION
  USE localstring:stringsize
  -> @localstring
  @localstring; outchar()
END -> @outstring
```

In fact any value can be assigned to an object. The right hand side of the "destacking" operation must be an address, hence the "@" notation. For example :

```
"A character string
containing line control <CR> characters" -> @string
1234->@constantvalue
```

4.1.1 Object Introduction

Syntax :

```
<object introduction> ::=
     USE { <intro> { <intro>}0:* }1:* .

<intro> ::=
     <name> { : <size> }

<size> ::=
     <name> | <numeric constant>

<name> ::=
     <alpha-numeric name> |
     <special name>

<alpha-numeric name> ::=
     <letter> { <letter>|<digit>}0:11

<letter> ::=
     A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
     a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
```

```
<digit> ::=
     0|1|2|3|4|5|6|7|8|9

<special name> ::=
     { <special character> }1:12

<special character> ::=
     !|"|£|$|%|&|'|(|)|*|:|[|{|}|]|^|?|/|>|<|.|+|-|=|@
```

Mule objects may be introduced in an object introduction. They are introduced with the keyword USE and are given a name. The name must be unique to the whole Mule environment. The name may be either a more normal alpha-numeric name, ie a string of characters commencing with an alphabetic character, or a special name. Special names are composed entirely of what are conventionally "operator" type characters and are intended to allow the provision of enhanced operations, should this be required.

The effect of the object introduction is to create an object table record and reserve data memory cells for it.

In addition to being given a name the object may be given a size attribute which specifies the number of data memory cells the object will occupy. If the size attribute is omitted 4 memory cells will be reserved for the object.

An object may be used without being introduced in which case the object is treated as if it had been introduced at the point of use and is given the default size attribute of 4.

Restrictions:
An object introduction may not appear within a function stacking operation which is itself nested within a function stacking operation.

An object name must be unique to the whole Mule environment.

An object cannot be introduced with a size attribute of zero.

Only 12 characters are significant in an object name. Any characters above the 12th are ignored.

Examples of use:

Valid object introductions are :

```
USE element  character:1  buffer:132.
USE char2:ele1.
```

Introduces
element which occupies 4 (default) memory cells.
character which occupies 1 memory cell.
buffer which occupies 132 memory cells.

buffer2 which occupies a variable number of memory cells which is dependent upon the value of ele1. If the value of ele1 is 12 when the USE is executed then 12 cells will be reserved.

```
USE  ::   <>  () : 12.
```

Note the need for syntactic separation between () and :.

Introduces
- :: which occupies 4 (default) memory cells.
- <> which occupies 4 (default) memory cells.
- () which occupies 12 memory cells.

Invalid object introductions are :

```
USE 1a  <abc>  abcdef:0  &&&.
```

Because
- 1a commences with a digit
- <abc> mixes special characters and alphabetic characters.
- abcdef:0 allocates a size attribute of zero.
- &&&. is a valid special name but the "." to signify the end of the introduction would be missing. There must be a space between the last special character and the "." terminator.

4.1.2 Stacking Operation

Syntax:

```
<stacking operation> ::=
     <name> |
     @ <name> |
     <constant> |
     <function call operation> |
     <function stacking operation>

<constant>  ::=
     <numeric constant> |
     <character string constant>

<numeric constant> ::=
     <integer constant> |
     <decimal constant> |
     <floating constant>

<integer constant> ::=
     { <digit> }0:* |
     H: { <hex digit> }1:* |
     O: { <octal digit> }1:* |
     B: { <binary digit> }1:*
```

```
<hex digit> ::=
     <digit> |A|B|C|D|E|F

<octal digit> ::=
     0|1|2|3|4|5|6|7

<binary digit> ::=
     0|1

<decimal constant> ::=
     {<digit>}1:* . {<digit>}0:*

<floating constant> ::=
     {<digit>}1:* E {<digit>}0:2

<character string constant> ::=
     " {<letter>|<digit>|<special character>|
     <line formatting characters>}0:* "
```

Semantics :
The appearance of a name, a constant or a function stacking operation will cause a value to be placed upon the stack. The value placed on the stack will depend upon the type of stacking operation and the size attribute of any object being stacked.

Objects which are being stacked consecutively maybe separated by commas or line formatting characters. Line formatting characters are spaces, tabulation characters and new line characters.

Taking each in turn :

name - the appearance of a name will cause the contents of the memory cells reserved for the object to be copied to the stack. Each cell will be copied in turn and the stack top will assume the size attribute of the original object. The contents of the memory cells allocated to the named object will be unchanged by the stacking operation.

@ name - the @ symbol indicates that the address of the first data memory cell of the named object is to be placed on the stack. The stack assumes a default size attribute (ie 4).

numeric constant - the numeric constant is placed on the stack and the stack assumes a defult size attribute. The precision used for decimal and floating constants is not defined here.

character string constant - the character string is placed on the stack. One memory cell is assigned to each character and the stack assumes a size attribute which is equal to the length of the character string. The character set representation is ASCII.

function stacking operation - see section 4.1.4.

Examples of use :

```
element, 123, "Character
               String which includes
               New line and space characters", H:FFEB3,
<> , :: , ::= , @element, @ <>
```

4.1.3 Function Call Operation

Syntax:

```
<function call operation> ::=
     ()
```

Semantics:
The function call operation will expect to find a function for it to execute on the top of the stack. The effect of the operation is to execute the function and, once done, continue execution at the next Mule operation following the function call operation.

The function will be placed upon the stack by a function stacking operation, or a stacking operation. Functions and the function stacking operations are described in section 4.1.4.

Examples of Use :

```
FUNCTION element, "Element Handler" END ()
```

... causes a function to be placed on the stack (ie all between the FUNCTION and END) and that function to be executed. The execution causes the value of element and the character string "Element Handler" (in turn) to be placed on the stack.

4.1.4 Function Stacking Operation

Syntax:

```
<function stacking operation> ::=
     FUNCTION <limited Mule language> END

<limited Mule language> ::=
     { <object introduction> |
     <limited stacking operation> |
     <function call operation> |
     <limited function stacking operation> |
     <limited infix stacking operation> |
     <destacking operation> }0:*

<limited stacking operation> ::=
     <name> |
     @ <name> |
     <constant> |
     <limited function stacking operation>

<limited infix stacking operation> ::=
```

```
        ( <limited expression> <operation>
        <limited expression> )

    <limited expression> ::=
        <limited stacking operation>
        {<function call operation>}0:* |
        <limited infix stacking operation>

    <limited function stacking operation> ::=
        FUNCTION {
        <limited stacking operation> |
        <function call operation> |
        <limited function stacking operation> |
        <limited infix stacking operation> |
        <destacking operation>
        } END
```

Semantics:
A function is a collection of Mule operations bracketed by the keywords FUNCTION and END. The effect of the function stacking operation is to place the address of the start of the function (in program memory) on the stack. The stack then assumes a size attribute of 4 memory cells.

The function is called by the application of a function call operation [ie ()]. The result of this is to cause execution of the first operation within the function bracket and to continue from there. When execution meets the function return, execution continues from the operation following the function call which caused the function to be executed.

A function (or strictly a function reference) may be assigned to a mule object by a destacking operation, see section 4.1.6.

Restrictions :
A function may contain within it a nested function stacking operation however, the nested function stacking operation may not contain object introductions. This restriction applies to explicit (ie USE type) and implicit (ie the appearance of an object name) introductions.

Examples of use :

```
FUNCTION
    char
END
```

... Execution of this function will cause the value of char to be placed on the stack.

```
FUNCTION FUNCTION char END END
```

... Execution of this function will cause a function reference to be placed on the stack. If this function is then executed then the value of char will be placed on the stack.

4.1.5 Infix Stacking Operation

Syntax:

```
<infix stacking operation> ::=
     ( <expression> {<operator> <expression>}0:1 )

<expression> ::=
     <name> {<function call operation>}0:* |
     @ <name> |
     <constant> |
     <function stacking operation>
          {<function call operation>}0:* |
     <infix stacking operation>

<operator> ::=
     <name>
```

Semantics:
The infix stacking operation is short hand for a function call which takes its parameters from the stack and delivers its result to the stack. For example :

```
( a + b ) is exactly equivalent to
a, b, + ()
```

The operator is any name which has been introduced either explicitly or implicitly and which has had a function reference assigned to it.

Restrictions :
The operator value must be a function reference.

Examples of use :

```
(a+b)
(f() AND (a OR b) )
( FUNCTION a,b END() TRIADICOPERATION c)
(x())
```

4.1.6 Destacking Operation

Syntax:

```
<destacking operation> ::=
     -> <destacking expression>

<destacking expression> ::=
     <name> |
     @ <name> |
     <constant> |
     <infix stacking operation>
```

Semantics:
The destacking operation causes the value which is currently on the stack to be transfered to the data memory address which is provided as the right hand operand to the destacking operation.

The address may be one of :

@<name> - in which case the value is assigned to the memory cells reserved for the object.

<name> - in which case an address is obtained from the value of the object named. The object must be 4 memory cells in size. The first 3 memory cells (cells 0 to 3) provide the address in data memory whilst memory cell 4 provides the size attribute of the object being referenced. An assignment of the form :

```
@a -> @b
```

will create a suitable reference such that cells 0 to 3 of the object "b" will contain the cell address of "a", whilst cell 4 of "b" will contain the value 4 which is the size attribute of "a".

<constant> values, except for character strings, assume the default size attribute of 4 memory cells. The constant, if used as a data memory address must provide the address in cells 0 to 3 and the size attribute in cell 4. As cell bit size is not specified as part of the Mule language the use of numeric constants in this way will make programs non-portable. Character strings may be used and portability maintained as each character occupies a memory cell and the character set is specified.

An <infix stacking operation> which must deliver a value which is an address reference which includes the size attribute, as described above.

Restrictions:
Except for constant values placed on the stack the size attribute of the stack value must be identical to that of the destack operand.

If the stack value is a constant then :

- If the constant has a larger size attribute than the destack operand then the constant is truncated to the same size as the operand and the higher order memory cells are discarded.

- If the constant has a smaller size attribute than the destack operand then the constant is copied to the lower order memory cells of the operand and the higher order memory cells are left unchanged.

Examples of use :

```
3 -> @a
```

... The value 3 is assigned to the lower 4 memory cells of the object "a".

"The rain in Spain" -> @str
... The character string "The rain in Spain" is assigned to the object "str" provided the object has a size attribute of 17 cells.

(a+ (b*c)) -> (index ARRAY @arrayname)
... Assigns the result of (a+ (b*c)) to an address evaluated by the expression (index ARRAY @arrayname)

4.1.7 Built-in Functions

Mule provides a number of built-in functions which are part of the kernel Mule environment. Although they are built-in they are invoked in exactly the same manner as normal Mule functions, ie by the function call operation [ie ()].

The built-in functions expect their arguments to be on the stack and the arguments must be valid for the uses specified. Arguments are removed from the stack by the operation of the function. Results are delivered to the stack.

4.1.7.1 Input/Output Functions

select — Has one argument which is a value which represents an input/output channel. Channels are represented by a number between 0 and 100.

The function selects the designated channel and any subsequent input or output is directed to that channel.

inchar — Has a single argument which is a data memory address. The function reads "n" characters from the currently selected input/output channel and transfers them to data memory at the address specified in its argument. The value of "n" is given by the size attribute of the data memory address of the argument.

outchar — Has a single argument which is a data memory address. The function copies "n" characters from data memory, at the address specified in its argument, to the currently selected input/output channel. The value "n" is given by the size attribute of the data memory address of the argument.

outint — Has a single argument which is the result of an expression involving numeric values. The function expects a value with a size attribute of 4. The function will interpret the lower order 4 cells as strict binary with cell 0 providing the least significant bits and cell 3 providing the most significant. The argument is transferred to the currently selected input/output channel as a decimal representation of the binary value.

4.1.7.2 Arithmetic Functions

Arithmetic functions have two arguments which represent numeric values. The functions expect values with size attributes of 4 and either an integer, decimal or floating format. In the case of integer arguments the function will interpret the lower order 4 cells as strict binary with cell 0 providing the least significant bits and cell 3 providing the most significant.

In the descriptions which follow the top of the stack is the second argument and the next stack item is the first argument. The function removes both arguments from the stack.

+	The first argument is added to the second argument and the result is placed on the stack.
-	The second argument is subtracted from the first argument and the result is left on the stack.
*	The first argument is multiplied by the second argument and the result is left on the stack.
/	The first argument is divided by the second argument and the result left on the stack. In the case of integer division the remainder is discarded and the result truncated to the integer result.
^	The first argument is raised to the power of the second argument and the result left on the stack.

4.7.1.3 Relational Functions

Relational functions have two arguments. Each argument may be multi-celled and may have dissimilar size attributes. The relational operator compares each memory cell in turn, treating the memory cell as strict binary with cell 0 representing the lower order cell. Where one argument has a greater size attribute the higher order memory cells of the lower size argument are padded with cells of value binary zero until both arguments are of identical size.

In the descriptions which follow the top of the stack is the second argument and the next stack item is the first argument. The function removes both arguments from the stack.

In the relational functions the first arument is compared with the second. The result is a constant value which is placed on the stack. The constant is either 1 (for a true relationship) or 0 (for a false relationship).

The functions are :

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal
/=	Not equal

4.1.7.3 Logical Functions

Logical functions have two arguments. Each argument may be multi-celled and may have dissimilar size attributes. The logical operation acts upon each memory cell in turn, treating the memory cell as strict binary with cell 0 representing the lower order cell. Where one argument has a greater size attribute the higher order memory cells of the lower size argument are padded with cells of value binary zero until both arguments are of identical size.

In the descriptions which follow the top of the stack is the second argument and the next stack item is the first argument. The function removes both arguments from the stack.

AND — The AND function acts upon each bit of its arguments in turn such that bit 0 of the first argument is ANDed with bit 0 of the second argument, and then bit 1, and so on.

OR — The OR function acts upon each bit of its arguments in turn such that bit 0 of the first argument is OR'd with bit 0 of the second argument, and then bit 1, and so on.

4.1.7.4 Miscellaneous Functions

IF — The IF function has two arguments which are on the stack. The second argument is the top stack item which is a function reference. The first argument is the second stack item which is a single celled value which is either 0 or 1. This is typically produced by the action of a relational or logical function or a combination of the two in an infix operation.

The action of the IF function is to call its second argument as a function if the first argument is the value 1. If the value is not 1 no action is taken.

In either case the function removes its arguments from the stack.

POP The POP function has no arguments. The action of the function is to remove a return address from the function return list. The effect of this is such that at the next function return, execution will return, not to the point of call of the currently executing function, but to the point of call of the function which called the currently executing function.

STACKNAME

The STACKNAME function has a single argument which is on the top of the stack. The argument is a reference to an introduced object. The result of the function is to place on the stack a character string constant which exactly matches the name of the object, eg

```
@convoy STACKNAME()
```

will put the character string "convoy" on the stack.

COMPILE The function COMPILE has one argument which is a character string constant and is on the top of the stack. The action of the COMPILE function is to stop reading from the terminal and take its input from a file which is given by the function's argument. The file name format is host machine dependent. Reading returns to the user's terminal if an error is found in the file being read, or the end of file is reached.

SUBSTR The SUBSTR function has 3 arguments as follows :

- The 3rd argument is on the top of the stack and is a numeric value which gives the length of a substring.

- The 2nd argument is next on the stack and is a numeric value which represents the start position of a substring. The first cell is cell zero.

- The first argument is next on the stack and is a reference to an object.

The action of the function is to obtain a reference to a portion of free space which is between cell 0 and cell-n of the object referenced by the first argument, where n is the size attribute (minus 1) of the object.

For example :

```
USE buffer:20.
@buffer 3,7 SUBSTR()
```

will place on the stack a reference to the free space location (@buffer + 3) with a size attribute of 7.

VALUE

The VALUE function has a single argument which is on the top of the stack. The argument represents a data memory reference. The function causes the contents of the memory cells referenced to be copied to the stack top.

For example :

```
USE buffer:20.
"ABC123DEF456GHI789JK" -> @buffer
@buffer 3,7 SUBSTR() VALUE()
```

will result in the character string value "123DEF4" being placed on the stack.

4.2 The Pilot Implementation

The Mule language was implemented in CORAL 66 on an ICL 2966 running the George 3 Operating System under DME. The implementation was a pilot to test out the basic ideas and had the following restrictions :

- Only upper case characters were permitted (a George 3 restriction).

- Memory cells were 6-bits (again a George 3 restriction, 8 bits would have been better).

- The size attribute in an object introduction was restricted to being a constant value.

- Only integer numeric values were implemented.

- Only decimal notation integer constants were implemented.

- In relational and logical functions the arguments were assumed to be no greater than 4 memory cells and higher order cells were discarded.

- The built in function "SELECT" was not implemented and all input output was directed to channel-0.

- The separators "," and ";" were considered to be identical to space and line separator characters and were completely interchangeable. In addition comments were permitted at any place where a separator was permitted. Comments were enclosed in square brackets (ie[...]) and could not contain either open or close square bracket characters.

The pilot implementation will be described in the following sections in terms of : An overview, Lexical phase and the interpretation phase.

4.2.1 Overview

The Mule Pilot Implementation (Mule-P) relies upon 2 stages, namely the lexical analysis (and syntax checking) phase and the interpretation phase.

Normally control passes to and fro between the phases as Mule-P source code is read. For example suppose a user entered :

```
( 5 * 3 ) OUTINT()
```

The lexical analysis phase would be called to decide that this was an infix stacking operation and to provide the value 5. The interpretation phase would then be called to cause the value 5 to be placed upon the stack. The lexical analysis phase would then read on and provide the value 3. The interpreter would be called

again to cause 3 to be stacked. The lexical analyser would then provide the multiplication function call which the interpreter would execute, and so on.

This to and fro operation is the normal mode of the Mule-P and allows fully interactive use with the user, for example, printing the values of objects for debugging should execution cease for any reason.

When the lexical analyser meets a function stacking operation it does not allow the interpeter to execute the function immediately but instead stores the body of the function in program memory and only calls the interpreter to stack the function reference.

In order for this mechanism to operate the lexical analyser and interpreter must share some common data structures.

The first is the linked object table and object string table which provides information about introducd objects.

The object table consists of records of the following structure referenced by an object table mark. Each record is identical.

- The size of the object, restricted to a range of 0 to 255. This is represented by the name "OBJ SIZE M".

- An index into the object string table, restricted to a range of 0 to 32767. This is represented by the name "OBJ STRING MARK M".

- The data memory address of the object. This is represented by the name "OBJ ADDRESS M"

The object string table is a character array represented by the name "OBJECT STRING DG", each element being 6 bits. The object string table is indexed by "OBJ STR MARK M" of the object table.

Starting at the index "OBJ STR MARK M" the characters have the following meaning :

Character-0 The length of the objects name in characters, eg the name "CONVOY" has a length of 6.

Character-1 to Character-n
The characters of the objects name.

Character-(n+1) to (n+3)
An index into the object table to provide the record for the object. The least significant bits are in character-(n+1).

The next data structure is data memory which is an array of 6-bit cells represented by the name "MEMORY DG". The data structure is indexed by "MEM MARK IG".

Program memory is represented by the data structure "PROGRAM DG" which is an array of 6-bit cells. Program memory is indexed by the variable "PROG MARK IG". Program memory consists of operator cells and operand cells. Operator cells are executed by the Mule-P interpreter, the value of the operator dictating the operation which is to be performed. Operator cells are structured as follows :

- Bits 4 to 5 represent the operation field which takes on the following values.

  0 - denotes that operand cells follow the operator.

  1 - denotes that no operand cells follow the operator.

- Bits 0 to 3 represent an operation modifier field which is used in conjunction with the operation field to denote the operation which is to be performed, ie :

Operation Field	Operation Modifier Field	Operation to be performed
0	1	Stack a value.
0	2	Stack an address.
0	3	Stack a literal (non character string constant).
0	4	Stack a character string literal (constant).
0	5	Declare an object.
1	1	Function call.
1	2	A direct branch causing the interpreter to index the program memory at a particular index.
1	3	A Function Return.
1	4	A destack.

Operand cells appear immediately following, in program memory, operator cells with operation modifier fields of the value "1". The value provided as an operand will depend upon the preceding operator as follows :

Operator	Operand
Stack a value	3 memory cells interpreted as strict binary with cell-0 providing the lower order bits and cell-2 providing the higher order bits.

	The value provided is the index into the Object Table for the object value to be stacked.
Stack an address	ditto.
Stack a literal	As the only literal value which is implemented is the integer the value of the integer is represented in 3 program memory cells with cell-0 providing the lower order bits and cell-2 providing the higher order bits.
Stack a character string literal	The number of memory cells of the operand is dependent upon the number of characters in the character string, ie the string length. If n is the length of the string then n+1 program memory cells form the operand. The first memory cell provides the length of the character string. Note that, with 6-bit memory cells, a character string is restricted to 64 characters.
Declare an object	As stack a value.

The last data structure is the interpreter's run-time stack. The interpreter interacts with the stack during stack, destack and operations which take their arguments from the stack, ie the function call and direct branch. The stack is represented by 2 arrays, both indexed by the variable "STACK MARK IG" which, unless in the process of being updated, always references the top stack item. The arrays are "STACK DG" which holds the stack operand and "STACK TYPE DG". Each element of "STACK DG" is 24-bits, ie 4 memory cells and is interpreted as a binary value using 2s complement notation. The meaning of the value of "STACK DG" is dependent upon the value of "STACK TYPE DG", as follows :

STACK TYPE DG	Meaning	STACK DG
0	Stack a value	The index into the object table for the object to be stacked. When the object is required its value will be obtained from the data memory address referenced.
1	Stack an address	The index into the object

		table for the object address to be stacked. When the object is required its data memory address will be obtained from the data table.
2	Constant	The number of elements of STACK DG used will depend upon the size attribute of the constant. STACK DG[STACK MARK IG] provides the size attribute of the constant. The cells which make up the constant are stored 4 to each element of STACK DG. Any unused space within an element of STACK DG is discarded.
3	Address Constant	Is stored as a 4-cell constant.

The Mule-P implementation is modular with CORAL 66 procedures forming the unit of modularity. Each procedure is compiled separately and communicates with other procedures through the data structures described above, through parameters and returned results and through global variables. Global variables are defined in a separate modules as are macros which were used to make changes to the code simpler.

In addition variable naming follows a convention with each variable having a 2 letter suffix. The first letter denotes usage and can be one of :

B	Boolean having the value "TRUE M" or "FALSE M"
I	Integer
K	Loop Counter
P	Procedure
D	Data structure (eg array or table)
Q	Procedure parameter
L	Label
M	Macro

The second letter denotes scope and can be either :

L	Local to the procedure or block.
G	Global to the whole program.
C	Common (declared in a CORAL 66 Common

communicator).

Execution starts with the call of procedure "LEXIS PG", the lexical analyser.

4.2.2 Lexical Analyser

Lexical Analysis (and syntax analysis) starts with the procedure "LEXIS PG".

4.2.2.1 LEXIS PG

LEXIS PG is called repetitively to read Mule-P source code. By calling other procedures it translates the source code into operations and operands which it places sequentially in program memory, PROGRAM DG. The sequence of events is :

If there are operations to interpret in PROGRAM DG, ie LEXIS PG is operating in interactive dialogue mode, the procedure INTERPRET PG is called to execute them. The program memory marker PROG MARK IG is reset to its initial value after execution.

The procedure FETCH ELEMENT PG is called to fetch an "element" of Mule-P. It returns the element in 2 global variables ELEMENT IG and ELEMENT TYPE IG. The meanings of the various options are as follows :

Value of ELEMENT TYPE IG	Value of ELEMENT IG	Meaning
ELE OPERAND M	Index into the data table of the object	The element was an object name, eg CONVOY, +, -> and has an entry in the object table.
ELE LITERAL M	The value of the literal (integers only are allowed)	The element was a constant value, eg 32, 123.
ELE CHAR LIT M	A pointer to the start of the character string which includes the string length as its first character.	The element was a character string, eg "A character string".

No other element type is permitted and an error message is produced, via the procedure ERRORHANDLER PG, if the element is not one of the above.

The error message is :

SYNTAX ERROR - UNKNOWN ELEMENT

If the element is an operand the action is dependent upon the value of ELEMENT IG, ie the index into the object table. Mule-P symbols are introduced via LEXIS PG as normal objects during system initialisation. As a result they are given specific object table indices as follows :

Data Table Index	Mule Symbol
0	:
2	.
4	USE
6	(
8	FUNCTION
10	->
12	()
14	@
16	END
18	)
20	'
22	"

Note that object table indices increase by 2 - this is a function of the way that the object size, object string mark and object address is mapped onto the data table array.

If the value of ELEMENT IG is < 19 (and ELEMENT TYPE IG is ELE OPERAND M) the action taken by LEXIS PG is dependent upon the value of ELEMENT IG, as follows :

ELEMENT IG=0, ie :
This is a development feature and can be used to signify the end of an element to FETCH ELEMENT PG.

ELEMENT IG=2, ie .
The . denotes the end of introductions which is handled by the procedure INTRODUCTION PG. LEXIS PG takes no action and passes control to the exit point - see later.

ELEMENT IG=4, ie USE
The procedure INTRODUCTION PG is called to handle object introductions. Control is then passed to the exit point - see later.

ELEMENT IG=6, ie (
The procedure INFIX PG is called to handle the infix operation. Control is then passed to the exit point - see later.

ELEMENT IG=8, ie FUNCTION
The variable REMAIN IN LEXIS KL is incremented to indicate to the exit point that control is to remain within LEXIS PG until the matching function-end is received. The procedure FUNCTION DEF PG is called to deal with the function

definition. Control is then passed to the exit point - see later.

ELEMENT IG=10, ie ->
The procedure DESTACK PG is called to handle the destack operation. Control is then passed to the exit point - see later.

ELEMENT IG=12, ie ()
The procedure FUNCTION CALL PG is called to handle the function call operation. Control is then passed to the exit point - see later.

ELEMENT IG=14, ie @
FETCH ELEMENT IG is called to fetch the next Mule-P element which should be an operand for the @ operator to act upon. The procedure STACK PG is then called to plant the stack address operation in program memory. Control is then passed to the exit point - see later.

ELEMENT IG=16, ie END
The variable REMAIN IN LEXIS KL is decremented to counter the increment at the corresponding FUNCTION. The procedure END FUNCTION DEF PG is called to handle the operation. Control is then passed to the exit point - see later.

ELEMENT IG=18, ie)
LEXIS PG should never receive a closed bracket as this is handled by INFIX PG. There is a small design error in this part of the procedure as LEXIS PG ignores any closed bracket received, passing control to the exit point. An error message should have been produced.

Any other value of ELEMENT IG, with ELEMENT TYPE IG having the value ELE OPERAND M, represents a user introduced object which is handled by the procedure STACK PG.

Similarly elements with ELEMENT TYPE IG of ELE LITERAL M or ELE CHARLIT M are handled by the procedure STACK PG.

STACK PG causes the appropriate stacking operations, and operands, to be planted in the program memory for interpretation by INTERPRET PG. Control is then passed to the exit point.

The exit point of LEXIS PG performs a check on the value of the variable REMAIN IN LEXIS PG. LEXIS PG will exit only if the value of this variable is 0. LEXIS PG is called recursively from INFIX PG and DESTACK PG and when called for Mule-P operations like :

```
(FUNCTION (A+ FUNCTION X Y Z() END() ) END() + B) ->
( FUNCTION (A<B) FUNCTION @A END FUNCTION @B END IFELSE() )
```

It is required that LEXIS PG handle the Mule-P between the

FUNCTION and END at the same recursion level.

4.2.2.2 INTRODUCTION PG

This procedure is called whenever an explicit introduction is met in the Mule-P source, eg USE A:10.

The procedure sets the flag IN INTRO BG to TRUE M. This is used by the procedure FETCH ELEMENT PG to indicate that it has been called from within INTRODUCTION PG.

Control remains in INTRODUCTION PG until the "." character is read to indicate the end of an introduction when the procedure returns control to LEXIS PG after first resetting the flag IN INTRO BG to FALSE M.

The following sequence is repeated for each object being introduced.

An element is obtained from the Mule-P source by calling FETCH ELEMENT PG. If the element obtained is not an operand, ie ELEMENT TYPE IG is not equal to ELE OPERAND M, the error handler procdure, ERRORHANDLER PG, is called. The error message generated is :

SYNTAX ERROR - UNKNOWN ELEMENT

The operand must be a new name being introduced. The introduction of the object, planting of the declare operation in program memory and creation of the object table record is all performed by the procedure FETCH ELEMENT PG as is necessary when the object is implicitly introduced (ie by using a name). It is FETCH ELEMENT PG which checks for duplicate introductions of the same object name. Allocation of data memory is done dynamically during interpretation.

FETCH ELEMENT PG returns the object table index allocated to the introduced object in ELEMENT IG. This is saved in a local variable.

FETCH ELEMENT PG is called again to check for a ":", in the case of :

USE A:12.

If the next element is not a ":" control returns to the head of the repeated sequence but without reading an element as this has been obtained already.

In the case of a ":" being read FETCH ELEMENT PG is called again to read the numeric constant which follows. If a numeric constant is not read ERRORHANDLER PG is called to print the error message :

SYNTAX ERROR - A NUMERIC CONSTANT MUST BE SPECIFIED AS AN ATTRIBUTE OF THE OBJECT INTRODUCTION.

When FETCH ELEMENT PG introduced the object it was given a default size attribute of 4 memory cells. This is now updated to the value specified in the actual introduction by updating the field OBJ SIZE M of the object table record.

4.2.2.3 INFIX PG

INFIX PG is called at the opening bracket of an infix stacking operation and control remains within INFIX PG until the matching close bracket is received. INFIX PG is called recursively to handle nested infix stacking operations.

INFIX PG causes the infix stacking operation to be translated into two stacking operations and a function call, ie (A AND B) is translated into A B AND(). The only complication is that the operands A and B can be very complex involving infix stacking operations, function calls and function stacking operations.

INFIX PG operates as follows :

The procedure LEXIS PG is called (recursively) to handle the first operand. This will cause operations to be planted in the program memory which will allow the interpreter to, ultimately, provide the first operand from the stack. Calling LEXIS PG in this way will cater for all possible types of operands and, in the case of an infix stacking operation as the first operand, INFIX PG will be called again, and recursively, from INFIX PG.

Having obtained the operand FETCH ELEMENT PG is called to obtain the infix operator (which is a Mule-P object name!). However it is possible that what is obtained is either (or a combination) :

- A function call symbol { () } in the case of (FUNCTION A B END() TRIADIC B END), or multiple function call symbols, eg in the case of (FUNCTION A B + END() () AND C).

  If this is the case the procedure FUNCTION CALL PG is called to plant the function call operation in the program memory for each occurance of the function call symbol. FETCH ELEMENT PG is then called to obtain the actual infix operator.

- A closed bracket in the case of a single operand, eg (X), or (FN()). In this case a return from INFIX PG to its calling procedure is performed.

INFIX PG checks that the operation name is valid, ie that ELEMENT IG = ELE OPERAND M and that the name is not the "@" symbol as this is invalid.

Once the actual infix operator name is obtained it is saved in a local variable.

LEXIS PG is called again to obtain the right hand operand. The

next element is fetched, by FETCH ELEMENT PG, and identical action taken to that above, in the case of the element being either a function call or closed bracket.

Finally STACK PG is called to plant operations in the program memory to allow the interpreter to stack the infix operator name. Then the procedure FUNCTION CALL PG is called to handle the planting of the function call operator.

A closed bracket, ie), is expected but it is possible that a further infix operation name and associated operand(s) may be obtained, eg (A + B * C - D). In this case INFIX PG repeats the infix operator name and right hand operand handling for each operator/operand.

4.2.2.4 FUNCTION DEF PG

FUNCTION DEF PG is called whenever the start of a function stacking operation is encountered in the Mule-P source (ie FUNCTION). The procedure END FUNCTION DEF PG is called to service the corresponding end.

The Mule-P implementation restricts the nesting level of function stacking operations to the value of FUNC STC SIZE M. If this is exceeded ERRORHANDLER PG is called to deliver the error message :

NESTED FUNCTION DEPTH EXCEEDED

Both FUNCTION DEF PG and END FUNCTION DEF PG communicate information through an integer array FUNCTION START DG. This is indexed by FUNC NEST DEPTH IG, the current nesting depth, which is incremented by FUNCTION DEF PG and decremented by END FUNCTION DEF PG. FUNCTION START DG is used to hold an index into program memory.

FUNCTION DEF PG plants the following in the program memory at the index which is put into FUNCTION START DG.

- Firstly a stack literal "0" operation. This will be changed by END FUNCTION DEF PG to a constant which is the index into program memory of the end of the function.
- Secondly a direct jump operation.

The effect of these operations will be to cause the interpreter to branch over the operations of the function until they are called by a function call operation.

4.2.2.5 FUNCTION CALL PG

This procedure is called when a function call operation, ie (), appears in the Mule-P source. It plants a function call operation into program memory and returns.

4.2.2.6 END FUNCTION DEF PG

END FUNCTION DEF PG is called when a function stacking operation end, ie END, is obtained.

It is possible that a function end has been entered without a corresponding FUNCTION symbol. If this is the case ERRORHANDLER PG is called to display the error message :

FUNCTION END ENCOUNTERED WITHOUT A FUNCTION BEGIN

Before returning END FUNCTION DEF PG does the following :

Plants a function return operation in program memory.

Changes the value of the stack literal operation at the function begin to the current program memory index (ie PROG MARK IG) from its initial value of zero.

Calls the procedure STACK PG to plant operations in program memory to cause the interpreter to stack the index into program memory of the start of the function.

Decrements the value of FUNC NEST DEPTH IG.

4.2.3 Interpreter

The lexical analyser, LEXIS PG, calls the interpreter, ie the procedure INTERPRET PG, whenever there are any operations in the program memory for it to interpret. Control remains in the interpreter until it meets a halt-operation within program memory when the interpreter returns control to LEXIS PG. Before returning, INTERPRET PG plants halt instructions in program memory for that portion of the program memory which represents interactive dialogue between the user and Mule-P. This is to prevent the operations from being re-executed by LEXIS PG on return.

INTERPRET PG controls its execution firstly by using the value of the operation field and then by using the value of the operation modifier. This double switch enables INTERPRET PG to obtain the operation's operand via a central handler rather than obtaining the operand for each operation.

In this way control is passed to particular parts of INTERPRET PG dependent upon the operation which is to be performed and in particular the following sub-routines are provided :

- Halt.
- Stack a value.
- Stack an address.
- Stack a literal.
- Stack a character string literal.
- Declare an object.
- Function call.
- Direct branch.
- Function return.
- Destack.

Built-in functions (see section 4.1.7) are part of the Mule-P interpreter but are called in the Mule-P source like any other function and the programmer is unaware that the function is built-in. The interpreter recognises a built-in function by its address field (ie OBJ ADDRESS M) of the data table record of the built-in function object being negative. The absolute value of this address is the built-in function number which is used by the interpreter. Built-in function numbers are assigned to introduced objects by a built-in function MAKEBUILTIN which has a preset function number of 1. Suppose we wished to create a built

in function name ADD with a function number of 4 (ie the same as the "+" function). The function call :

```
ADD,4 MAKEBUILTIN()
```

Will set the address field (OBJ ADDRESS M) of the data table record of the object ADD to -4 signifying that the function is built-in (ie the address is negative) and that the function number is 4. The function number is used as a parameter to the procedures BUILT IN PG which is called by INTERPRET PG whenever a built-in function is to be called. BUILT IN PG has CORAL 66 routines for each of the built-in functions of section 4.1.7. Control of execution within BUILT IN PG is by using the function number passed as a parameter to the procedure.

Function numbers are as follows :

Function Number	Function Name
1	MAKEBUILTIN
2	INCHAR
3	OUTCHAR
4	+
5	-
6	*
7	/
8	^ (exponentiation)
9	<
10	>
11	<=
12	>=
13	=
14	<>
15	AND
16	OR
17	(unallocated - reserved for exclusive OR)
18	IF
19	GOTO
20	POP
21	STACKNAME
22	OUTINT
23	COMPILE
24	SUBSTR
25	VALUE

4.3 Test Programs

At the outset of the design of the Mule-P implementation 3 tests were devised :

- Test A to test the basic operation of introductions and to be operating prior to implementation of any other operation.

- Test B to test the infix stacking operation (and relying upon Test A operating successfully).

- ACCEPTANCE which tests all of the features of the Mule-P language and introduces 2 mule functions, via the COMPILE built-in function, IFELSE which provides the IF...THEN...ELSE capability and WHILE which provides the WHILE...DO....ENDDO facility.

Mule-P passed all of these tests.

4.4 Conclusions

The Mule-P implementation is compact and operates in approximately 18k words (around 64k bytes) which includes trace code used during debugging. The whole environment was written in CORAL 66 with the interpreter, which is the only part which needs to be resident at all times, being approximately 1000 lines of code (inclusive of comments). The IFELSE and WHILE functions were very easy to produce and showed that the Mule-P environment can be enhanced very easily. A programmer would be unable to tell that the IFELSE and WHILE functions were not part of the Mule-P kernel. Other facilities, eg an editor, could be added in the same manner.

For better structure to programs it would be useful to add the concept of mode or type to Mule-P. This could be by, say, the addition of a function TYPE to the built-in repertoire to assign type to an introduced object and CHECK to check for correct use of the type. For example :

```
USE integerobject, referenceobject.
     [ Introduces two objects which may be of any type]

integerobject, integer TYPE()
referenceobject, reference TYPE()
     [ Restricts "integerobject" to be of type "integer"  and
     "referenceobject" to be of type "reference" ]

FUNCTION
     USE p1,p2.
     integer CHECK() -> @p1
     integer CHECK() -> @p2
     (p1 + p2)
END -> @ ++
     [Introduces  a  function  called  "++"  which  adds two
```

operands which it finds on the stack but checks that they are "integer" type operands before allowing the operation]

```
(integerobject ++ 3) -> @integerobject
     [is legal as "integerobject" is an integer]
(referenceobject ++ 3) -> @referenceobject
     [is illegal as "referenceobject" is a reference]
```

In order for this type of mechanism to operate successfully it is necessary to bar access to the non-typed versions of the functions, ie + in the above example. This could be achieved by adding access control to Mule. For example if two functions CLOSE and OPEN were added the following would be possible:

```
+ ++ OPEN()
     [Open access to the "+" function from the "++"
     function, ie allow the "++" function to call the "+"
     functon]
+ CLOSE()
     [Close all other accesses to the "+" function. Now
     only the "++" function can call the "+" function so
     that to open access again the "++" function would need
     to contain the OPEN function call]
OPEN CLOSE()
     [Is lethal and closes all access to the OPEN function]
```

It is thought that only the above additions are necessary to the Mule kernel to make it a truely useful environment for transporting programs. The syntax of the language would require improving but it was not considered that the syntax was important in proving the concept of the Mule-P implementation.

A final improvement would be the implementation of the lexical analyser procedures in Mule and translating them to the Mule operations in program memory. If this was achieved it would be necessary to transport the interpreter (some 1000 lines) only when moving the Mule environment.

4.5 Mule-P Source Code

The Appendix contains the Coral 66 source code of the Mule-P implementation, the George 3 command language instructions for compiling and operating the Mule-P and the Mule-P source code for building the environment and for testing.

The following are the files included and their uses :

File Name	Purpose
PCOR(/MACR)	The CORAL 66 compilation macro used to invoke the compiler. It uses another George 3 macro (PINCLUDE) to include CORAL 66 source files

in sequence for the compiler. Compilation ceases if errors are found in a particular source file. This is because the compiler's error handling is very poor.

PINCLUDE(/MACR)
A George 3 macro used with PCOR to present each CORAL 66 source file in a sequence to the compiler.

PCON(/MACR) The linking, or "consolidation" George 3 maco which brings together the compiled Mule source and the standard input/output libraries.

PMULE(/MACR) The George 3 macro which executes the Mule-P.

The following are the Mule-P source files.

HEADER Compiler introduction statements.

COMMON Declaration of common objects and those which will be used from the standard subroutine library.

MIDDLE CORAL 66 required statements between the Common comunicator and the start of the procedure declarations.

MACROS Macros definitions.

VARIABLES Global variable declarations

The following are Mule-P procedures.

TRACEPC Used for debugging.

ERRORHANDLER ERRORHANDLER PG

READARELCHAR READ A REL CHAR PG

FETCHELEMENT FETCH ELEMENT PG

STACKPG STACK PG

FUNCTIONCALL FUNCTION CALL PG

INFIXPG INFIX PG

ENDFUNCTIOND END FUNCTION DEF PG

FUNCTIONDEFP FUNCTION DEF PG

INTRODUCTION INTRODUCTION PG

POPVALUEPG	POP VALUE PG
POPADDRESSPG	POP ADDRESS PG
POPINTEGERPG	POP INTEGER PG
BUILTINPG	BUILT IN PG
INTERPRETPG	INTERPRET PG
MAPPG	MAP LIST PG - used for providing a map of memory addresses for each global variable. Used for debugging.
LEXISPG	LEXIS PG
END	The start of executable code and compilation end statements.

The following are Mule-P source files.

LANGSYMB(/MULE)	Contains a list of the Mule-P symbols and built in routines which area read into the Mule-P at system initialisation.
BUILTINS(/MULE)	Contains the calls of MAKEBUILTIN which allocates built in function numbers to the built in functions introduced above. This file is executed immediately following system initialisation.
TESTA(/MULE)	Mule TEST-A.
TESTB(/MULE)	Mule TEST-B.
IFELSE(/MULE)	The if..then...else function.
WHILE(/MULE)	The while function.
ACCEPTANCE(/MULE)	The Mule-P acceptance test.

5 REFERENCES

1 Backus J 1978
Can programming be liberated from the Von Neuman Style. A functional style and its algebra of programs.
CACM V21 N8.

2 Lucas P 1970
Lauer P
Sfigleitner H
Method and notation for the formal definition of programming languages.

3 Brown P J (Ed)
Software Portability - An advanced course
Publisher - Cambridge University Press

4 HMSO
The Official Definition of Coral 66

5 Warren J C 1976
Software Portability - a survey of approaches and problems.
10th IEEE Computer Society International Meeting on Technology to Reach the People.

6 Richards M 1971
The portability of the BCPL complier.
Software Practice and Experience V1 pp135-146

7 Coleman S S 1974
Poole P C
Waite W M
The MOBILE programming system JANUS.
Software Practice and Experience V4 pp5.

8 Capon P C 1972
Wilson I R
Morris D
Rohl J C
The MUS compiler target language and autocode.
Computer Journal V15

9 Snyder A et al 1975
A portable language for the language C
MIT MAC-TR-149

10 Colin A J T et al 1975
The translation and interpretation of STAB-12.
Software Practice and Experience V5 N2 pp 123-138.

11 Johnston S C
A portable compiler - theory and practice.
Bell Laboratories, New Jersey.

12 Koster C H A 1974
Portable compilers and the Uncol problem.
Machine Oriented High Level Languages.
North Holland Publishing Co, Amsterdam.

13 Poole P C 1971
Hierarchical abstract machines.
Proceedings of the Culham Symposium on Software Engineering.
HMSO

14 Poole P C 1970
Waite W M
Building Mobile Software.
Computer Journal V13 N28

15 Orgas R J 1969
Waite W M
A base for a mobile programming system.
CACM V12 N11

16 Brown P J 1972
Levels of language for Portable Software.
CACM 15.

17 Abrhams P S 1971
An APL machine - PhD Thesis
Report NB SLAC 114 Stamford California

18 McCracken D D 1973
Revolution in Programming
Datamation December 1973.

19 Dijkstra E W 1972
The Humble Programmer
CACM V15 N10 1972 pp 859-866

20 Newey M C 1972
Poole P C
Waite W M
Abstract machine modelling to produce portable software.
Software Practice and Experience V2 pp 107-136.

21 Dunn R C 1973
SNOBOL 4 as a language for bootstrapping a compiler.
Sigplan Notices V8 pp28-32

22 Steel T B 1961
The first version of Uncol.
Procs IFIPS WJCC V19.

23 L Star - An interactive, symbolic implementation system.
CMU Pittsburgh.
NTIS AD-A050 119/7GA

24 Koster C H A 1974
Portable Compilers & the Uncol Problem.
Machine Oriented Higher Level Languages.
North Holland Publishing Company, Amsterdam.

25 Taylor J R 1977
A compiler interpreter for a LISP dialect.
Research Establishment RISO Report; Denmark.

26 Brown P J 1976
Throw away compiling.
Software Practice and Experience.

27 Wilcox T R 1971
Generating Machine Code for High Level Programming Languages.

28 Bauer F L 1975
(Editor)
Lecture notes in Computer Science 30 : Software Engineering and Advanced Course.

29 Ammann U 1977
On Code Generation in a Pascal compiler.
Software Practice and Experience V7 N3 pp391-423

30 Lecharm O et al 1974
A (Truely) Usable & Portable Compiler Writing System.
Information Processing 74, pp218-221
North Holland Publishing Company, Amsterdam.

31 Miller P L 1971
Automatic Creation of a Code Generator from a Machine Description.
Project MAC TR-85 (May 1971)

32 Lauer P E 1977
Abstract Tree Processors with netwoeks of state machines as control - their use in programming language definition.
Colloquium on Trees in Algebra and in Programming, 5-7 Feb 1976, pp96-128 (Lille - France).

33 1978
Computer Software - Transferability & Portability; Bibliography with Abstracts.
NTIS PS-78 00552 6GA

34 The Structure of Directly Executed Languages - A New Theory of Interpretive System Design.
Stamford University California - Digital Systems Lab.
NTIS AD-A051 835 7GA

35 Lecrame O 1973
An Experience in Structured Programming and Transferability.

ACM Sigplan Notices V8 N9 Sep 73 pp956.

36 Cheriton Et al 1978
Thoth, A Portable Real Time Operating System.

37 Allen J
The anatomy of Lisp.
Publisher - McGraw Hill.

38 Chevance R J 1977
Design of High Level Language Oriented Processors.
Sigplan Notices V12 N11, pp40-51

39 Chung L L 1977
Elements of Discrete Mathematics.

40 Wilson R J 1972
Introduction to Graph Theory.
Publisher - New York Academic Press.

41 McCarthy Et Al 1965
Lisp 1.5 Programmers Manual
MIT Press

42 Akin D C 1978
Allen B C
Simulation of the Direct Execution of Higher Order Languages
NTIS AD-A055 235 6GA

APPENDIX

The Appendix contains the Coral 66 source code of the Mule-P implementation, the George 3 command language instructions for compiling and operating the Mule-P and the Mule-P source code for building the environment and for testing.

The following are the files included and their uses :

File Name	Purpose
PCOR(/MACR)	The CORAL 66 compilation macro used to invoke the compiler. It uses another George 3 macro (PINCLUDE) to include CORAL 66 source files in sequence for the compiler. Compilation ceases if errors are found in a particular source file. This is because the compiler's error handling is very poor.
PINCLUDE(/MACR)	A George 3 macro used with PCOR to present each CORAL 66 source file in a sequence to the compiler.
PCON(/MACR)	The linking, or "consolidation" George 3 maco which brings together the compiled Mule source and the standard input/output libraries.
PMULE(/MACR)	The George 3 macro which executes the Mule-P.

The following are the Mule-P source files.

HEADER	Compiler introduction statements.
COMMON	Declaration of common objects and those which will be used from the standard subroutine library.
MIDDLE	CORAL 66 required statements between the Common comunicator and the start of the procedure declarations.
MACROS	Macros definitions.
VARIABLES	Global variable declarations

The following are Mule-P procedures.

TRACEPC	Used for debugging.
ERRORHANDLER	ERRORHANDLER PG

READARELCHAR	READ A REL CHAR PG
FETCHELEMENT	FETCH ELEMENT PG
STACKPG	STACK PG
FUNCTIONCALL	FUNCTION CALL PG
INFIXPG	INFIX PG
ENDFUNCTIOND	END FUNCTION DEF PG
FUNCTIONDEFP	FUNCTION DEF PG
INTRODUCTION	INTRODUCTION PG
POPVALUEPG	POP VALUE PG
POPADDRESSPG	POP ADDRESS PG
POPINTEGERPG	POP INTEGER PG
BUILTINPG	BUILT IN PG
INTERPRETPG	INTERPRET PG
MAPPG	MAP LIST PG - used for providing a map of memory addresses for each global variable. Used for debugging.
LEXISPG	LEXIS PG
END	The start of executable code and compilation end statements.

The following are Mule-P source files.

LANGSYMB(/MULE)	Contains a list of the Mule-P symbols and built in routines which area read into the Mule-P at system initialisation.
BUILTINS(/MULE)	Contains the calls of MAKEBUILTIN which allocates built in function numbers to the built in functions introduced above. This file is executed immediately following system initialisation.
TESTA(/MULE)	Mule TEST-A.
TESTB(/MULE)	Mule TEST-B.
IFELSE(/MULE)	The if..then...else function.

WHILE(/MULE) The while function.

ACCEPTANCE(/MULE)
The Mule-P acceptance test.

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.PCOR(32/MACR) PRODUCED ON  1DEC81 AT 19.34.52

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.54.47 USING I381

DOCUMENT    PCOR(32/MACR)


    0 IF MOP,RP FB,CM,CT,LG,OJ
    1 WE COMERR,EX
    2 IF NOT EXI(CMULE),CE CMULE(*DA,KWOR15,BUCK1)
    3 £LO :LIB.PROGRAM CORA
    4 LOAD :UTILITY.BCORAL66=325
    5 EN
    6 AS *DA7,CMULE(WRITE)
    7 AS *LPU,ERRORS
    8 PINCLUDE HEADER,%A
    9 RL *LPU
   10 AS *LPU
   11 PINCLUDE COMMON,%A
   12 PINCLUDE MIDDLE,%A
   13 PINCLUDE MACROS,%A
   14 PINCLUDE VARIABLES,%A
   15 PINCLUDE TRACEPC,%A
   16 PINCLUDE ERRORHANDLERPG,%A
   17 PINCLUDE READARELCHARPG,%A
   18 PINCLUDE FETCHELEMENTPG,%A
   19 PINCLUDE STACKPG,%A
   20 PINCLUDE FUNCTIONCALLPG,%A
   21 PINCLUDE INFIXPG,%A
   22 PINCLUDE ENDFUNCTIONDEFPG,%A
   23 PINCLUDE FUNCTIONDEFPG,%A
   24 PINCLUDE INTRODUCTIONPG,%A
   25 PINCLUDE DESTACKPG,%A
   26 PINCLUDE POPVALUEPG,%A
   27 PINCLUDE POPADDRESSPG,%A
   28 PINCLUDE POPINTEGERPG,%A
   29 PINCLUDE BUILTINPG,%A
   30 PINCLUDE INTERPRETPG,%A
   31 PINCLUDE MAPPG,%A
   32 PINCLUDE LEXISPG,%A
   33 IF MOP,RP FB,CM,LG,CT
   34 PINCLUDE END,%A
   35 1PCOR
   36 DP 0,LAST FILE COMPILED WAS : %X
   37 IF DEL(FI£XPCK),(DP 0,COMPILED OK)ELSE(DP 0,          .*****                    COMPILED WITH ERRORS )
   38 IF COR,DL
   39 IF NOT DEL(FI£XPCK),LF ERRORS,FR2
   40 IF MOP,RP FB,CM
   41 EX

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

Â*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.PINCLUDE(6/MACR) PRODUCED ON 1DEC81 AT 19.40.04

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.59.53 USING I381

DOCUMENT   PINCLUDE(6/MACR)

```
SP A,(%A)(1,12)
RL *LPU
SP X,REPLY(21,21)
IF NOT STR(%X)=(0),GO 1PCOR
IF ABS(*SH),DP 0,%A
£AL 772,-2
AL 774,-2
AS *TRO,%A
IN ERRORS,T$$$$,PS


COMPILED %A WITH ERRORS
$$$$
AS *LPO,ERRORS(APPEND)
SP W,(X)
SP aW,(%A)
RM
SP V,REPLY
SP aW,(%A-%V)
IF NOT FAILED(FILE *TRO EXHAUSTED),GO 1PCOR
EXIT
```

Â***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
Ā*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZH.PCON(5/MACR)  PRODUCED ON  8APR81 AT 13.04.58

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.54.41 USING I381

DOCUMENT    PCON(5/MACR)

      0 IF MOP,REPORT FB,CM,LG
      1 IF EXISTS(LINKMAP),ER LINKMAP
      2 CE !
      3 IN !,T$$$$
      4 *IN (CMULE)
      5 *LIB ED (IOLIB)
      6 *LIB ED (SUBGROUPCORA)
      7 *LIST
      8 ****
      9 $$$$
     10 CONSOLIDATE *TR!,*LPLINKMAP,INCMULE,IN:LIB.IOLIB,IN:LIB.SUBGROUPCORA
     11 ER !
     12 AS *LP1,LINKMAP(APPEND)
     13 EN 7
     14 SP X,CORE
     15 SP X,VALUE(%X+1)
     16 AL 27,%X
     17 SV BMULE
     18 DL
     19 IF MOP ,REPORT FB,CM
     20 EX

Ā***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

Ā*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZN.PMULE(7/MACR) PRODUCED ON 31MAR82 AT 17.34.34

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.57.51 USING I381

DOCUMENT   PMULE(7/MACR)

```
LF :R P S.OU.PMULE,FRS/MULE/,LI1
RP FB,CM,LG,OJ,CT
IF STR(%A)=(),GO 1BYPASS
IF NOT EXI(%A),GO 1ERROR
1BYPASS
LO :R P S.OU.BMULE
AS *CR0,:R P S.OU.LANGSYMB
AS *LP0
EN
AS *CR0,:R P S.OU.BUILTINS
EN 1
IF PRE(*TA),ON %(*TA)
IF PRE(*LI),(AL 29,%(*LI))ELSE(AL 29,0)
AS *CR0,%A
EN 1
1RETURN IF HAL(IN),GO 1IN
IF HAL(ST),GO 1ST
RP FB,CM
EX
1IN SP X,DISPLAY
AS *CR1,%X
RM
EN 2
GO 1RETURN
1ERROR LF :R P S.OU.PMULE,FRS/ERROR 1/,LI1
RP FB,CM
EX
1ST LF :R P S.OU.PMULE,FRS/STOPPED/,LI1
RP FB,CM
EX
MULE V1 - DEFECT FIX LEVEL 0
ERROR 1 - MULE FILE PASSED AS A PARAMETER DOES NOT EXIST
STOPPED TO ALLOW DEBUG - CORE IMAGE & PERIPHERALS PRESERVED.  TYPE RM TO CONT
```

Ā***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**

£LISTING OF :ZZZZZZZZZZZN.HEADER(1/)    PRODUCED ON  8APR81 AT 13.05.16

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.57.18 USING I381

DOCUMENT    HEADER(1/)

      0 'PROGRAM' MULE
      1 'ERRORLIST'
      2 'SENDTO'(CMULE,C)
      3 'LEADERS'
      4 ****

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.COMMON(15/)   PRODUCED ON 20JAN82 AT 17.40.15

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.57.21 USING I381

DOCUMENT    COMMON(15/)

'COMMON'
'PROCEDURE' DISPLAY ('VALUE''INTEGER');
'PROCEDURE' ERROR HANDLER PG('VALUE''INTEGER','VALUE''INTEGER');
'PROCEDURE' BACKSPACE;
'INTEGER''PROCEDURE' READ;
'INTEGER''PROCEDURE' READCHAR;
'PROCEDURE' WRITETEXT('VALUE''INTEGER');
'PROCEDURE' PRINT('VALUE''INTEGER');
'PROCEDURE' SPACES('VALUE''INTEGER');
'PROCEDURE' NEWLINE;
'PROCEDURE' PRINTCH('VALUE''INTEGER');
'PROCEDURE' SELECTOUT('VALUE''INTEGER');
'PROCEDURE' SELECTIN ('VALUE''INTEGER');
'PROCEDURE' LEXIS PG;
'PROCEDURE' INFIX PG;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**~

£LISTING OF :ZZZZZZZZZZZZN.MIDDLE(2/)    PRODUCED ON  8APR81 AT 13.05.27

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.58.48 USING I381

DOCUMENT    MIDDLE(2/)

       'PROGRAM' MULE
       'BEGIN'
       'INTEGER' KEEP COMPILER SWEET ;

~***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

MACROS(52/)

```
DEFINE' ELE LITERAL M"0" ;
DEFINE' ELE OPERAND M"1" ;
DEFINE' TA PROC M"0" ;
DEFINE' EXIT M"'GOTO' EXIT LL" ;
DEFINE' ICHAR M"READCHAR" ;
DEFINE' NEW LINE M"-1" ;
DEFINE' FALSE M"0" ;
DEFINE' RELAVANT M(Q)"Q<>16 'AND' Q<>-1 'AND' Q<>28 'AND' Q<>11 'AND' Q<>'LITERAL'([)" ;
DEFINE' TRUE M"1" ;
DEFINE' REPEAT M(Q)"'GOTO' Q" ;
DEFINE' BACK STEP M"BACKSPACE" ;
DEFINE' NAME LENGTH M"62" ;
DEFINE' ALPHABETIC M(Q)"Q>='LITERAL'(A) 'AND' Q<='LITERAL'(Z)" ;
DEFINE' NUMERIC M(Q)"Q>='LITERAL'(0) 'AND' Q<='LITERAL'(9)" ;
DEFINE' INC M(Q1,Q2)"Q1:=Q1+Q2" ;
DEFINE' NON ALPHANUMERIC M(Q)"Q>=10 'AND' Q<=32" ;
DEFINE' RETURN M"'GOTO' RETURN LL" ;
DEFINE' EMPTY M"63" ;
DEFINE' DEFAULT OBJ SIZE M"4" ;
DEFINE' BYTE 2 M(Q)"'BITS'[6,12]Q" ;
DEFINE' BYTE 1 M(Q)"'BITS'[6,6]Q" ;
DEFINE' BYTE 0 M(Q)"'BITS'[6,0]Q" ;
DEFINE' OBJ TABLE SIZE M"1000" ;
DEFINE' OBJ STR SIZE M"10000" ;
DEFINE' PROG SIZE M"10000" ;
DEFINE' RAT DECLARE M"5" ;
DEFINE' OBJ SIZE M"'BITS'[8,16]OBJ TAB DG" ;
DEFINE' OBJ STR MARK M"'BITS'[16,0]OBJ TAB DG" ;
DEFINE' OSEL M(Q)"SELECT OUT(Q)" ;
'DEFINE' OMAPF M"1" ;
'DEFINE' PTRACE M(Q1,Q2) "TRACE PC(Q1,Q2)";
'DEFINE' ONL M "NEWLINE" ;
'DEFINE' OSP M(Q) "SPACES(Q)" ;
'DEFINE' OTEXT M(Q) "WRITETEXT(Q)" ;
'DEFINE' OINT M(Q) "PRINT(Q)";
'DEFINE' ICH M "READ CHAR";
'DEFINE' OCH M(Q) "PRINTCH(Q)";
'DEFINE' RAT COLON M "0";
'DEFINE' RAT DOT M "2";
'DEFINE' ISEL M(Q) "SELECT IN(Q)";
'DEFINE' IINITF M "1";
'DEFINE' STOP CHAR M "'LITERAL'(?)";
'DEFINE' RAT END M"16";
'DEFINE' RAT BRANCH M"18";
'DEFINE' RAT ST VAL M"1";
'DEFINE' RAT ST ADD M"2";
'DEFINE' RAT ST LIT M"3";
'DEFINE' RAT AT M"14";
'DEFINE' RAT CRD M"18";
'DEFINE' RAT ORD CRD M "12";
'DEFINE' RAT FN CALL M"17";
'DEFINE' VAL HELD M"0";
'DEFINE' ADDR HELD M"1";
'DEFINE' LIT HELD M"2";
'DEFINE' MEM SIZE M "10000";
'DEFINE' FUNC STC SIZE M "25";
'DEFINE' STACK SIZE M "200";
'COMMENT'(***************************************************************
THIS MACRO HAD TO BE COMMENTED OUT & MADE INTO A PROCEDURE BECAUSE OF THE
INABILITY OF THE COMPILER TO HANDLE IT - SEE PROCEDURE TRACE PC
'DEFINE' UPSTACK M "STACK MARK IG := STACK MARK IG+1;
   'IF' STACK MARK IG>STACK SIZE M 'THEN'
   'BEGIN'
        ERROR HANDLER PG(6,""RUN%TIME%ERROR"");
   'END'";
END OF COMMENTED OUT SECTION
*************************************************************);
'DEFINE' RAT DESTACK M"20";
'DEFINE' RAT FN RET M "19";
'DEFINE' ELE CHAR LIT M "2";
'DEFINE' CHAR LIT 1 DELIMITER M "18";
'DEFINE' CHAR LIT 2 DELIMITER M "23";
'DEFINE' CHAR LIT LENGTH M "63";
'DEFINE' RAT ST CHAR LIT M "4";
'DEFINE' ISEL M(Q) "SELECTIN(Q)";
'DEFINE' IF TTY M "0";
'DEFINE' IF COMPILE M "1";
'DEFINE' MULE MESSAGE M (Q1,Q2) "DISPLAY(Q2);'CODE''BEGIN' SUSWT('LIT'(Q1))'END'";
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZN.VARIABLES(28/)    PRODUCED ON  4MAR82 AT 17.04.12

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.53.59 USING I381

DOCUMENT    VARIABLES(28/)


      0 'INTEGER'
      1 ELEMENT IG,  'COMMENT' AN ELEMENT OF MULE ;
      2 ELEMENT TYPE IG,  'COMMENT' THE TYPE OF ELEMENT READ ;
      3 IRRELAVANT SKIPPED BG,  'COMMENT' A FLAG WHICH IS SET TO TRUE IF AN IRRELEVANT CHARACTER WAS READ DURING THE  -
            LAST READ;
      4 CURRENT LINE IG, 'COMMENT' THE LINE NUMBER IN THE FILE BEING READ;
      5 OBJ TAB MARK IG,  'COMMENT' WHERE WE ARE UP TO IN THE OBJECT TABLE ;
      6 MAX OBJ TABLE MARK IG,  'COMMENT' WHERE WE ARE UPTO (MAXIMUM) IN THE OBJECT TABLE;
      7 TA PROC PRINT IG, 'COMMENT' THE NUMBER OF TRACE MESSAGES OUTPUT ;
      8 LATEST PROCEDURE IG, 'COMMENT' THE LAST PROCEDURE CALLED ;
      9 PROG MARK IG, 'COMMENT' POINTER INTO PROGRAM MEMORY;
     10 KG, 'COMMENT' GLOBAL COUNTER - USE CAREFULLY TO AVOID CONCURRENT USE;
     11 FUNC NEST DEPTH IG,'COMMENT' THE STATIC NESTING LEVEL OF FUNCTION DEFINITIONS;
     12 RUN TIME PROG MARK IG, 'COMMENT' THE PROGRAM MEMORY MARKER DURING INTERPRETATION;
     13 START HERE IG,'COMMENT' THE PLACE TO START THE INTERPRETATION;
     14 STACK MARK IG, 'COMMENT' RUN TIME STACK MARKER;
     15 FUNC STC MARK IG, 'COMMENT' THE RUN TIME FUNCTION STACK MARKER;
     16 MEM MARK IG,'COMMENT' THE MOMORY MARKER;
     17 CHAR BUFF MARK IG, 'COMMENT' THE CHARACTER BUFFER MARKER ;
     18 IN INTRO BG, 'COMMENT' FLAG SET IF AN INTRODUCTION IS BEING PROCESSED;
     19 SIZE OF OBJECT IG, 'COMMENT' SIZE OF REFERENCED OBJECT RETURNED FROM "POPADDRESS PG" ;
     20 DUMMY
     21 ;
     22
     23 'COMMENT' PRESET ;
     24 'COMMENT'
     25 'INTEGER'
     26 :=
     27 ;
     28
     29 'BYTE' 'ARRAY'
     30 OBJECT STRING DG[0:OBJ STR SIZE M],
     31 PROGRAM DG[0:PROG SIZE M], 'COMMENT' PROGRAM MEMORY ;
     32 STACK TYPE DG[0:STACK SIZE M],
     33 MEMORY DG[0:MEMSIZE M],
     34 CHARACTER BUFFER DG[0:120], 'COMMENT' THE CHARACTERS INPUT BUFFER FOR THE CURRENT LINE;
     35 DUMMYA[0:1]
     36 ;
     37
     38 'INTEGER' 'ARRAY'
     39 OBJ TAB DG[0:OBJ TABLE SIZE M],  'COMMENT' THE OBJECT TABLE ;
     40 STACK DG[0:STACK SIZE M],
     41 FUNC MEMORY DG[0:FUNC STC SIZE M],
     42 FUNC STACK DG[0:FUNC STC SIZE M],
     43 FUNC NAME DG[0:FUNC STC SIZE M],
     44 FUNCTION START DG[0:FUNC STC SIZE M],
     45 DUMMYIA[0:1]
     46 ;
     47
     48 'COMMENT' PRESET;
     49 'INTEGER' 'ARRAY'
     50 POWER OF 10 DG[0:6] ,
     51 POWER OF 2 DG[0:15],
     52 DUMMYPIA[0:1]
     53 :=
```

```
      54 1,10,100,1000,10000,100000,1000000,
      55 1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
      56 0
      57 ;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

OF :ZZZZZZZZZZZZN.VARIABLES(28/)   PRODUCED ON 4MAR82 AT 17.04.12

BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.53.59 USING I381

VARIABLES(28/)

```
'INTEGER'
ELEMENT IG,  'COMMENT' AN ELEMENT OF MULE ;
ELEMENT TYPE IG,  'COMMENT' THE TYPE OF ELEMENT READ ;
IRRELAVANT SKIPPED BG,  'COMMENT' A FLAG WHICH IS SET TO TRUE IF AN IRRELEVANT CHARACTER WAS READ DURING THE
     LAST READ;
CURRENT LINE IG, 'COMMENT' THE LINE NUMBER IN THE FILE BEING READ;
OBJ TAB MARK IG,  'COMMENT' WHERE WE ARE UP TO IN THE OBJECT TABLE ;
MAX OBJ TABLE MARK IG,  'COMMENT' WHERE WE ARE UPTO (MAXIMUM) IN THE OBJECT TABLE;
TA PROC PRINT IG, 'COMMENT' THE NUMBER OF TRACE MESSAGES OUTPUT ;
LATEST PROCEDURE IG, 'COMMENT' THE LAST PROCEDURE CALLED ;
PROG MARK IG, 'COMMENT' POINTER INTO PROGRAM MEMORY;
KG, 'COMMENT' GLOBAL COUNTER - USE CAREFULLY TO AVOID CONCURRENT USE;
FUNC NEST DEPTH IG,'COMMENT' THE STATIC NESTING LEVEL OF FUNCTION DEFINITIONS;
RUN TIME PROG MARK IG, 'COMMENT' THE PROGRAM MEMORY MARKER DURING INTERPRETATION;
START HERE IG,'COMMENT' THE PLACE TO START THE INTERPRETATION;
STACK MARK IG, 'COMMENT' RUN TIME STACK MARKER;
FUNC STC MARK IG, 'COMMENT' THE RUN TIME FUNCTION STACK MARKER;
MEM MARK IG,'COMMENT' THE MOMORY MARKER;
CHAR BUFF MARK IG, 'COMMENT' THE CHARACTER BUFFER MARKER ;
IN INTRO BG, 'COMMENT' FLAG SET IF AN INTRODUCTION IS BEING PROCESSED;
SIZE OF OBJECT IG, 'COMMENT' SIZE OF REFERENCED OBJECT RETURNED FROM "POPADDRESS PG" ;
DUMMY
;

'COMMENT' PRESET ;
'COMMENT'
'INTEGER'
:=
;

'BYTE' 'ARRAY'
OBJECT STRING DG[0:OBJ STR SIZE M],
PROGRAM DG[0:PROG SIZE M], 'COMMENT' PROGRAM MEMORY ;
STACK TYPE DG[0:STACK SIZE M],
MEMORY DG[0:MEMSIZE M],
CHARACTER BUFFER DG[0:120], 'COMMENT' THE CHARACTERS INPUT BUFFER FOR THE CURRENT LINE;
DUMMYA[0:1]
;

'INTEGER' 'ARRAY'
OBJ TAB DG[0:OBJ TABLE SIZE M],  'COMMENT' THE OBJECT TABLE ;
STACK DG[0:STACK SIZE M],
FUNC MEMORY DG[0:FUNC STC SIZE M],
FUNC STACK DG[0:FUNC STC SIZE M],
FUNC NAME DG[0:FUNC STC SIZE M],
FUNCTION START DG[0:FUNC STC SIZE M],
DUMMYIA[0:1]
;

'COMMENT' PRESET;
'INTEGER' 'ARRAY'
POWER OF 10 DG[0:6] ,
POWER OF 2 DG[0:15],
DUMMYPIA[0:1]
:=
1,10,100,1000,10000,100000,1000000,
1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
0
;
```

```
ING OF :ZZZZZZZZZZZN.TRACEPC(18/)  PRODUCED ON 29JUL82 AT 17.04.26

UT BY LISTFILE IN ':TLFAK=10.RPSMITH' ON 16AUG82 AT 10.58.51 USING I381

IENT    TRACEPC(18/)


 0 'PROCEDURE' DUMMY PC; 'BEGIN' 'END' DUMMY;
 1
 2
 3 'PROCEDURE' TRACE PC ('VALUE' 'INTEGER' TRACE CODE Q, TRACE STRING Q );
 4 'BEGIN'
 5
 6 'COMMENT' TRACE PROCEDURE SWITCHED BY 'ON' OPERATIONS,IE
 7   ON 23 PROCEDURE TRACE
 8   ON 22 VALUE PRINT TRACE
 9   ON 21 STRING PRINT TRACE
10    'IF' TRACE CODE Q=TA PROC M 'THEN' LATEST PROCEDURE IG := TRACE STRING Q;
11
12
13    'IF' CURRENT LINE IG<[29]
14    'THEN' RETURN M;
15
16    'IF' ([30] 'MASK' POWER OF 2 DG[TRACE CODE Q]) 'DIFFER' POWER OF 2 DG[TRACE CODEQ]=0
17    'THEN'
18    'BEGIN'
19       'SWITCH' SELECT TRACE SL :=PROC LL,
20       VAL PRINT LL,
21       STRING LIST LL;
22       'GOTO' SELECT TRACE SL[TRACE CODE Q+1];
23
24    PROC LL:
25
26       'IF' TA PROC PRINT IG>7
27       'THEN'
28       'BEGIN'
29          TA PROC PRINT IG := 0;
30          ONL M;
31       'END';
32 OSP M(1);
33       OTEXT M(TRACE STRING Q);
34       INC M(TA PROC PRINT IG,1);
35       EXIT M;
36
37    VAL PRINT LL:
38
39       'IF' TA PROC PRINT IG > 4
40       'THEN' ONL M;
41       OTEXT M("£VALUE%IN%");
42       OTEXT M(LATEST PROCEDURE IG);
43       OSP M(1);
44       OINT M(TRACE STRING Q);
45       OCH M(TRACE STRING Q);
46       TA PROC PRINT IG := 10;
47       EXIT M;
48
49    STRING LIST LL:
50       ONL M;
51       OTEXT M("£STRING%IN%");
52       OTEXT M(LATEST PROCEDURE IG);
53       OSP M(1);
54       OTEXT M(TRACE STRING Q);
55       TA PROC PRINT IG := 10;
56       EXIT M;
57
58    EXIT LL:
59    'END';
60
61 RETURN LL:
62 'END'TRACE PC;
63
64
65 'COMMENT' THIS PROCEDURE SHOULD BE A MACRO BUT THE COMPILER COULDN'T
66   HANDLE IT;
67
68
69 'PROCEDURE' UPSTACK M;
70 'BEGIN'
71    INC M(STACK MARK IG,1);
72
73    'IF' STACK MARK IG>STACK SIZE M
74    'THEN' ERROR HANDLER PG(6,"STACK%OVERFLOW");
75 'END';

BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*
```

```
TING OF :ZZZZZZZZZZZN.ERRORHANDLER(15/) PRODUCED ON 24MAR82 AT 19.40.32

PUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.04.09 USING I381

MENT    ERRORHANDLER(15/)


'PROCEDURE' ERROR HANDLER PG('VALUE' 'INTEGER' ERROR SEVERITY Q,ERROR STRING Q);

'COMMENT'
PURPOSE
  OUTPUT ERROR MESSAGES TO THE USER IN THE FORMAT :
  ERROR NUMBER(FIRST PARAMETER),SPACE,ERROR STRING(SECOND PARAMETER)
MACROS
  OCH M
  OINT M
  ONL M
  OTEXT M
  OSP M
  ISEL M
  IF TTY M
GLOBAL VARIABLES
  CHARACTER BUFFER DG (R)
  CHAR BUFF MARK IG (R)
  RESTART LC (E)
LOCAL VARIABLES
  KL (LOOP COUNTER)
METHOD
  1. THE ERROR SEVERITY CAN BE W (WARNING), E (ERROR) OR R(RUN TIME ERROR)
  2. ACTION IS TAKEN APPROPRIATE TO THE SEVERITY
  3. W - MESSAGE GIVEN
     E - MESSAGE GIVEN & COMPILATION STOPPED
     R - MESSAGE GIVEN & EXECUTION STOPPED
;

'BEGIN'
   'INTEGER' KL;
   ONL M;

'COMMENT' THE ACTION HERE IS DEPENDENT UPON THE SEVERITY OF THE ERROR;


   'IF' ERROR SEVERITY Q = 'LITERAL'(W)
   'OR' ERROR SEVERITY Q = 'LITERAL'(E)
   'THEN'
   'BEGIN'

'COMMENT' A COMPILE TIME MESSAGE - PRINT THE CURRENT LINE BEING PROCESSED;

      OTEXT M("LINE%");
      OINT M(CURRENT LINE IG);
      OSP M(1);

      'FOR' KL := 0 'STEP' 1 'UNTIL' CHAR BUFF MARK IG+1
      'DO' OCH M(CHARACTER BUFFER DG[KL]);
      OTEXT M("%--%");
      OTEXT M(ERROR STRING Q);

      'IF' ERROR SEVERITY Q = 'LITERAL'(E)
      'THEN'
      'BEGIN'
         ISEL M(IF TTY M);
         'GOTO' RESTART LC ;
      'END';
   'END'
   'ELSE'
   'BEGIN'

      OTEXT M("RUN%TIME%ERROR");
      OSP M(1);OTEXT M("%--%");
OTEXT M(ERROR STRING Q);
ONL M;
'GOTO' RESTART LC;
      RUN TIME PROG MARK IG := PROG SIZE M ; (TEMPORARY MEASURE)
   'END';
   ONL M;
'END'ERROR HANDLER PG;

*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3
```

```
Ã*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.READARELCHAR(18/) PRODUCED ON 23JUN81 AT 19.00.20

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.02.26 USING I381

DOCUMENT    READARELCHAR(18/)


      0
      1
      2 'INTEGER' 'PROCEDURE' READ A REL CHAR PG;
      3
      4 'COMMENT'
      5 PURPOSE
      6   THIS PROCEDURE READS THE INPUT STREAM AND RETURNS ONLY "RELAVANT"
      7   CHARACTERS. RELAVANT CHARACTERS ARE ALL CHARACTERS BUT SPACES,
      8   NEWLINES..ETC,COMMAS AND SEMICOLONS. A NEWLINE CHARACTER
      9   RECEIVED CAUSES THE LINE COUNTER CHARACTER TO BE INCREMENTED. IF AN
     10   IRRELAVANT CHARACTER IS ENCOUNTERED IN THE INPUT STREAM AND
     11   SKIPPED OVER BY THIS ROUTINE THE GLOBAL FLAG "IRRELAVANT SKIPPED BG"
     12   IS SET TO "TRUE M". THIS IS NEEDED BECAUSE IRRELAVANT CHARACTERS
     13   ARE USED TO TERMINATE MULE OBJECT NAMES, EG "USE A" IS 2 NAMES
     14   WHEREAS "USEA" IS ONE NAME.
     15 GLOBAL VARIABLES
     16   IRRELAVANT SKIPPED BG (W)
     17   CURRENT LINE IG (R/W)
     18   CHARACTER BUFFER DG(W)
     19   CHAR BUFF MARK IG (RW)
     20 LOCAL VARIABLES
     21   CHAR IL (THE CHARACTER READ FROM THE INPUT STREAM)
     22 MACROS
     23   ICH M
     24   ONL M
     25   NEW LINE M
     26   FALSE M
     27   RELAVANT M
     28   TRUE M
     29   REPEAT M
     30   INC M
     31   STOP CHAR M
     32 RETURNS
     33   THE RELAVANT CHARACTER READ FROM THE INPUT STREAM
     34 METHOD
     35   1.THE FLAG IRRELAVANT SKIPPED BG IS SET TO FALSE M
     36   2.A CHARACTER IS READ FROM THE INPUT STREAM
     37   3.IF IT IS RELAVANT THE CHARACTER IS RETURNED
     38   4.IF IT IS IRRELAVANT THE FLAG "IRRELAVANT SKIPPED BG" IS SET TO
     39     TRUE M.
     40   5.A NEWLINE CHARACTER WHICH IS READ CAUSES THE CURRENT LINE COUNT TO
     41     BE INCREMENTED.
     42   6.ONLY RELAVANT CHARACTERS ARE RETURNED.
     43   7.COMMENTS, IE THOSE DELIMETED BY [..] ARE SWALLOWED
     44 ;
     45
     46 'BEGIN'
     47    'INTEGER' CHAR IL;
     48    PTRACE M(TA PROC M,"*");
     49    IRRELAVANT SKIPPED BG := FALSE M;
     50
```

```
    'THEN' CHAR BUFF MARK IG := -1;(JUST IN CASE - BY SOME FLUKE WE OVERRUN THE LINE)
    INC M(CHAR BUFF MARK IG,1);
    CHARACTER BUFFER DG[CHAR BUFF MARK IG] := CHAR IL;

'COMMENT' CAUSES AN INSTANT HALT TO OCCUR WHEN THE "STOP CHARACTER"
          IS READ.  THE REST OF THE LINE IS SWALLOWED SO THAT WHEN
          THE PROGRAM IS REENTERED THE BUFFERS ARE CLEAN (HOPEFULLY);


    'IF' CHARIL=STOP CHAR M
    'THEN'
    'BEGIN'

       'FOR' CHAR IL := ICH M 'WHILE' CHAR IL <> NEWLINE M
       'DO' ;
       ONL M;
       'CODE' 'BEGIN'
          SUSWT('LITERAL'(ST))
       'END';
    'END';

    'IF' RELAVANT M(CHAR IL)
    'THEN'
    'BEGIN'

       'IF' 'BITS'[1,0][30]=1 'AND' CURRENT LINE IG >= [29]
       'THEN' OCH M(CHAR IL);
       'ANSWER' CHAR IL
    'END'
    'ELSE'
    'BEGIN'
       IRRELAVANT SKIPPED BG := TRUE M;

       'IF' CHAR IL = NEW LINE M
       'THEN'
       'BEGIN'
          INC M(CURRENT LINE IG,1);
          CHAR BUFF MARK IG := -1;
       'END';

       'IF' CHAR IL = 'LITERAL'(E)
       'THEN'
       'BEGIN'

'COMMENT' MULE COMMENTS ARE DELIMITED BY [..]. THIS CODE SWALLOWS COMMENTS;

          'FOR' CHAR IL := ICH M 'WHILE' CHAR IL<>'LITERAL'(])
          'DO'
          'BEGIN'

             'IF' CHAR IL= NEW LINE M
             'THEN' INC M(CURRENT LINE IG,1);
          'END';
       'END';
       REPEAT M(FETCH CHAR LL);
    'END';
'END'READ A REL CHAR PG;
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
~*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.FETCHELEMENT(30/) PRODUCED ON  8APR82 AT 14.23.35

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.02.59 USING I381

DOCUMENT    FETCHELEMENT(30/)



'PROCEDURE' FETCH ELEMENT PG;

'COMMENT'
PURPOSE
  THIS PROCEDURE IS RESPONSIBLE FOR READING THE MULE INPUT STREAM
  AND TRANSLATING IT INTO 'ELEMENTS' OF MULE. ELEMENTS ARE COMBINATIONS
  OF CHARACTERS WHICH ARE GROUPED INTO NAMES. THE CHARACTER GROUPS ARE
  TERMINATED BY EITHER TABULATION CHARACTERS (EG SPACE, NEWLINE), COMMAS,
  SEMICOLONS, OR CHARACTERS OF A DIFFERENT ELEMENT TYPE. ELEMENT TYPES
  ARE :
    - AN ALPHABETIC CHARACTER FOLLOWED BY ALPHABETIC OR NUMERIC CHARACTERS
    - A COLLECTION OF NUMERIC CHARACTERS
    - A COLLECTION OF NON-ALPHABETIC AND NON-NUMERIC CHARACTERS
  COMPLETELY NUMERIC ELEMENTS REPRESENT LITERAL CONSTANTS (NOTE THIS
  PROCEDURE RESTRICTS THE MULE TO INTEGER CONSTANTS ONLY) WHICH ARE
  RETURNED INTO THE GLOBAL VARIABLES "ELEMENT IG" WHICH IS SET TO THE
  CONSTANT VALUE AND "ELEMENT TYPE IG" IS SET TO "ELE LITERAL M"
  TO DENOTE A CONSTANT VALUE.
  OTHER ELEMENTS REPRESENT "NAMES" (EG VALID NAMES ARE ANAME,
  NAME1,ANOTHERNAME,:^*-,!,&=/£)
  IN MULE IF AN OBJECT IS USED WITHOUT FIRST BEING INTRODUCED THE USE OF
  THE NAME AUTOMATICALLY INTRODUCES THE OBJECT.  THIS PROCEDURE DEALS WITH
  THE INTRODUCTION
GLOBAL VARIABLES
  OBJECT TABLE DG (R/W)
  FUNC NEST DEPTH IG (R)
  IN INTRO BG (R)
  OBJ TABLE MARK IG (R/W)
  OBJECT STRING DG (W)
  IRRELAVANT SKIPPED BG (R)
  POWER OF 10 DG (R)
  ELEMENT IG (W)
  ELEMENT TYPE IG (W)
  MAX OBJ TABLE MARK IG (R/W)
  PROGRAM DG (W)
  PROGRAM MARK IG (R/W)
LOCAL VARIABLES
  STRING DELIMITER IL (THE STRING DELIMITER USED BY THE USER)
  ELEMENT DL (ARRAY CONTAINS CHARACTERS OF ELEMENT BEING READ)
  CHAR IL  (CHARACTER READ FROM MULE STREAM)
  KL (LOOP CHARACTER)
  HASH IL (HASHING VALUE INTO THE OBJECT STRING TABLE )
MACROS
  OBJ SIZE M (W)
  OBJ STR MARK M (W)
  BACK STEP M
  NAME LENGTH M
  ALPHABETIC M
  NUMERIC M
  TRUE M
```

```
    ELE OPERAND M
    RETURN M
    EMPTY M
    DEFAULT OBJ SIZE M
    BYTE 2 M
    BYTE 1 M
    BYTE 0 M
    OBJ TABLE SIZE M
    OBJ STR SIZE M
    PROG SIZE M
    RAT DECLARE M
    ELE CHAR LITERAL M
    CHAR LIT 1 DELIMITER M
    CHAR LIT 2 DELIMITER M
    CHAR LIT LENGTH M
    ICH M
    NEW LINE M
  PROCEDURES CALLED
    READ A REL CHAR PG
    ERROR HANDLER PG
  MESSAGES PRODUCED
    2  TOO MANY NAMES INTRODUCED
    3  NO PROGRAM SPACE LEFT
  METHOD
    1. RELAVANT CHARACTERS (IE ALL CHARACTERS BUT SPACE, COMMA , SEMICOLON
       AND NEWLINE) ARE READ FROM THE MULE INPUT STREAM. AS THE READ
       PROCEDURE 'READ A REL CHAR PG' RETURNS THE RELAVANT CHARACTERS READ
       ONLY IT IS NECESSARY FOR IT TO SET THE FLAG 'IRRELAVANT SKIPPED BG'
       WHEN AN IRRELAVANT CHARACTER HAS BEEN SKIPPED IMMEDIATELY FOLLOWING
       A RELAVANT CHARACTER. THIS PERMITS NAMES TO BE TERMINATED BY AN
       IRRELAVANT CHARACTER WITHOUT "SEEING" THE IRRELAVANT CHARACTER.
    2. A MULE ELEMENT IS READ COMPLETELY AND PACKED CHARACTER BY CHARACTER
       INTO A LOCAL ARRAY 'ELEMENT DL' COMMENCING AT ELEMENT '1' OF THE
       ARRAY.  ELEMENT 0 OF THE ARRAY CONTAINS THE CHARACTER LENGTH OF THE
       ELEMENT.
    3. A NUMERIC ELEMENT OR A CHARACTER STRING-LITERAL WHICH IS READ IS "RETURNED".
    4. A NON-NUMERIC ELEMENT WHICH IS READ IS TREATED AS A "NAME". IF THE NAME
       IS KNOWN ALREADY TO THE MULE ENVIRONMENT (ENTRY EXISTS IN THE OBJECT
       STRING TABLE) ITS OBJECT TABLE INDEX IS "RETURNED". IF IT IS NOT
       KNOWN TO THE MULE ENVIRONMENT THE OBJECT IS INTRODUCED AND ITS OBJECT
       TABLE INDEX (NEWLY CREATED) IS "RETURNED".
;

'BEGIN'
   'INTEGER' 'ARRAY' ELEMENT DL[0:CHAR LIT LENGTH M];
   'INTEGER' CHARIL,KL,HASH IL,STRING DELIMITER IL;
   PTRACE M(TA PROC M,"FETCHELE");

'COMMENT' READ THE ELEMENT;

   CHARIL := READ A REL CHAR PG;
   ELEMENT DL[1]:= CHAR IL;
   KL := 2;

   'IF' ALPHABETIC M(CHAR IL)
   'OR' NUMERIC M(CHARIL)
   'THEN'
   'BEGIN'

'COMMENT' CASE OF ELEMENT IS AN ALPHA-NUMERIC NAME OR A LITERAL VALUE
   ;
```

```
121         'BEGIN'
122
123           'IF' IRRELAVANT SKIPPED BG = TRUE M
124           'THEN' 'GOTO' OUT1 LL;
125           ELEMENT DL[KL]:= CHAR IL;
126           INC M(KL,1);
127         'END';
128
129     OUT1LL:
130     'END'
131     'ELSE'
132     'BEGIN'
133
134 'COMMENT'
135               CASE OF ELEMENT IS AN "OPERATOR" TYPE NAME (IE NON ALPHA-NUM-NUMERIC)
136               OR A CHARACTER STRING LITERAL;
137
138
139 'COMMENT' CHECK FIRST FOR CHARACTER STRING LITERAL;
140
141         STRING DELIMITER IL := CHAR IL;
142
143         'IF' STRING DELIMITER IL = CHAR LIT 1 DELIMITER M
144         'OR' STRING DELIMITER IL = CHAR LIT 2 DELIMITER M
145         'THEN'
146         'BEGIN'
147
148 'COMMENT' THIS IS A CHARACTER STRING LITERAL;
149
150
151 'COMMENT' FETCH THE STRING AND RETURN IT;
152
153           KL := 1;
154
155           'FOR' CHAR IL := ICH M 'WHILE' CHAR IL <> STRING DELIMITER IL
156           'DO'
157           'BEGIN'
158              ELEMENT DL [KL] := CHAR IL ;
159              INC M(KL,1);
160
161              'IF' KL>CHAR LIT LENGTH M
162              'THEN' ERROR HANDLER PG('LITERAL'(E),"CHARACTER%STRING%LITERAL%IS%TOO%LONG");
163
164              'IF' CHAR IL = NEW LINE M
165              'THEN' INC M(CURRENT LINE IG,1);
166           'END';
167           ELEMENT DL[0] := KL-1; (CHARACTER STRING LENGTH)
168           ELEMENT TYPE IG := ELE CHAR LIT M;
169           ELEMENT IG := 'LOCATION' (ELEMENT DL[0]);
170           RETURN M;
171         'END';
172
173         'FOR' CHAR IL := READ A REL CHAR PG 'WHILE' NON ALPHA NUMERIC M(CHARIL)
174         'DO'
175         'BEGIN'
176
177            'IF' IRRELAVANT SKIPPED BG=TRUE M
178            'THEN' 'GOTO' OUT2LL;
179            ELEMENT DL[KL] := CHAR IL;
180            INC M(KL,1);
181         'END';
182
183     OUT2LL:
184     'END';
185     BACK STEP M;
```

```
187    ELEMENT DL[0] := KL-1;
188
189 'COMMENT' AT THIS STAGE THE ARRAY 'ELEMENT DL' IS SET UP SUCH
190          THAT ELEMENT-0 IS THE CHARACTER LENGTH OF THE ELEMENT AND ELEME
191          NT-1 TO
192          ELEMENT-N CONTAIN THE CHARACTERS OF THE ELEMENT;
193
194
195    'IF' NUMERIC M(ELEMENT DL[1])
196    'THEN'
197    'BEGIN'
198
199 'COMMENT' CASE OF A NUMERIC CONSTANT BEING RETURNED;
200
201       KL := ELEMENT DL[0]+1;
202       ELEMENT IG := 0;
203
204       'FOR' KL := KL-1 'WHILE' KL > 0
205       'DO'
206       'BEGIN'
207          ELEMENT IG := ELEMENT IG+POWER OF 10 DG[ELEMENT DL[0]-KL]*ELEMENTDL[KL];
208       'END';
209       ELEMENT TYPE IG := ELE LITERAL M;
210       RETURN M;
211    'END'
212    'ELSE'
213    'BEGIN'
214
215 'COMMENT' THIS IS THE COMPLEX PART OF THE PROCEDURE WHERE A "NAME" IS
216     TO
217             BE RETURNED. FIRSTLY IT IS NECESSARY TO SEARCH THE OBJECT TA
218             BLE TO SEE IF
219             THE NAME HAS BEEN INTRODUCED ALREADY;
220
221       HASH IL := 0; (WORK OUT THE HASHING VALUE - CURRENTLY NO HASH ONLY A
222    LINEAR
223      SEARCH)
224
225    SEARCH FOR NAME LL:
226
227       'IF' OBJECT STRING DG[HASH IL] <> EMPTY M
228       'THEN'
229       'BEGIN'
230
231 'COMMENT' COMPARE STRING TABLE WITH THE ELEMENT NAME STORED IN ELE
232       MENT DL;
233
234
235          'FOR' KL := 0,KL+1 'WHILE' KL <= OBJECT STRING DG[HASH IL]
236          'DO'
237          'BEGIN'
238
239             'IF' ELEMENT DL[KL] <> OBJECT STRING DG[KL+HASH IL]
240             'THEN'
241             'BEGIN'
242
243 'COMMENT' THE NAME IS NOT THE ELEMENT SO TRY THE NEXT NAME;
244
245                HASH IL := HASH IL+4+OBJECT STRING DG[HASH IL];
246                REPEAT M(SEARCH FOR NAME LL);
247             'END';
248          'END';
249
250 'COMMENT' THIS IS THE ELEMENT NAME SO FETCH THE OBJECT TABLE INDEX
```

```
253                 IG"TO
254                 "ELE OPERAND M";
255
256           HASH IL := HASH IL +OBJECT STRINGDG[HASH IL]+1;
257           ELEMENT IG := OBJECT STRING DG[HASH IL]*64*64+OBJECTSTRING DG[HASH IL+1]*64+OBJECTSTRINGDG[
258           HASH IL+2];
259           ELEMENT TYPE IG := ELE OPERAND M;
260
261           'IF' IN INTRO BG <> FALSE M 'AND' ELEMENT IG<>RAT DOT M 'AND' ELEMENT IG<>RAT COLON M
262           'THEN' ERRORHANDLER PG('LITERAL'(W),
263 "INTRODUCTION%OF%AN%OBJECT%WHICH%HAS%BEEN%INTRODUCED%PREVIOUSLY");
264       'END'
265       'ELSE'
266       'BEGIN'
267
268 'COMMENT' THIS IS THE CASE WHERE THE NAME IS NOT KNOWN AND HENCE T
269       HE
270                 OBJECT MUST BE DECLARED;
271
272
273 'COMMENT' CHECK THAT THE STATIC FUNCTION NESTING DEPTH IS NOT > 0
274 AS DECLARATIONS ARE NOT PERMITTED IF THIS IS THE CASE;
275
276
277           'IF' FUNC NEST DEPTH IG > 0
278           'THEN' ERROR HANDLER PG('LITERAL'(E),
279 "NAMES%CANNOT%BE%INTRODUCED%IN%A%NESTED%FUNCTION%DEFINITION");
280
281 'COMMENT' IF THIS IS NOT AN INTRODUCTION (IE A "USE ...") WARN THE
282 MAN THAT WE ARE INTRODUCING AN OBJECT;
283
284
285           'IF' IN INTRO BG = FALSE M
286           'THEN' ERROR HANDLER PG('LITERAL'(W),"INTRODUCTION%OF%AN%OBJECT%ASSUMED");
287
288 'COMMENT' A HASHING METHOD HAS NOT BEEN FORMULATED AS YET AND THER
289       EFORE
290                 THE CODE WHICH NEEDS TO ALTER THE VALUE OF HASH IL TO ENS
291                 URE THAT THE
292                 OBJECTS STRING DOES NOT CROSS A HASHING BOUNDARY IS NOT P
293                 RESENT. AT THE
294                 MOMENT A SIMPLE LINEAR SEARCH IS USED - THIS WILL BE CORR
295                 ECTED. 'HASH IL'
296                 NOW POINTS AT THE NEXT FREE OBJECT STRING TABLE LOCATION;
297
298           OBJ SIZE M[MAX OBJ TABLE MARK IG]:= DEFAULT OBJ SIZE M;(FILL IN OB
299       JECT SIZE)
300           OBJ STR MARK M[MAX OBJ TABLE MARK IG]:= HASH IL;(AND THE PLACE WHE
301       RE THE
302             NAME WILL BE KEPT AS AN INDEX INTO THE OBJECT STRING TABLE. USE
303       THE NEXT
304             FREE OBJECT TABLE SLOT)
305
306 'COMMENT' NOW PUT THE NAME OF THE OBJECT INTO THE STRING TABLE;
307
308
309           'FOR' KL := 0,KL+1 'WHILE' KL <= ELEMENT DL[0]
310           'DO' OBJECT STRING DG[HASH IL+KL]:= ELEMENT DL[KL];
311
312 'COMMENT' AND NOW LINK BACK TO THE OBJECT TABLE - INTO 3 BYTES WIT
313       H THE
314                 MOST SIGNIFICANT BYTE FIRST;
315
316           INC M(HASH IL,ELEMENT DL[0]+1);
```

```
319           OBJECT STRING DG[HASH IL+2] := BYTE 0 M(MAX OBJ TABLE MARK IG);
320
321  'COMMENT' PLANT CODE TO DECLARE THE OBJECT;
322
323           PROGRAM DG[PROG MARK IG] := RAT DECLARE M;
324           PROGRAM DG[PROG MARK IG+1] := BYTE 2 M(MAX OBJ TABLE MARK IG);
325           PROGRAM DG[PROG MARK IG+2] := BYTE 1 M(MAX OBJ TABLE MARKIG);
326           PROGRAM DG[PROG MARK IG+3]:= BYTE 0 M(MAX OBJ TABLE MARK IG);
327           INC M(PROG MARK IG,4);
328
329           'IF' PROG MARK IG > PROG SIZE M
330           'THEN' ERROR HANDLER PG('LITERAL'(E),"NO%PROGRAM%SPACE%LEFT");
331
332  'COMMENT' NOW SET UP THE RETURN VALUES - FIRSTLY "ELEMENT IG" IS S
333        ET TO
334                 THE OBJECT TABLE INDEX OF THE OBJECT AND "ELEMENT TYPE IG" IS SET
335                 TO "ELE OPERAND M";
336
337           ELEMENT IG := MAX OBJ TABLE MARK IG;
338           ELEMENT TYPE IG := ELE OPERAND M;
339
340  'COMMENT' AND FINALLY TIDY UP. FIRSTLY INCREMENT THE "MAX OB TABLEMARK IG"
341                 AND THEN CHECK FOR OVERFLOW OF THE OBJECT TABLE AND OBJEC
342                 T STRING TABLE;
343
344           INC M(MAX OBJ TABLE MARK IG,2);
345
346           'IF' MAX OBJ TABLE MARK IG > OBJ TABLE SIZE M
347           'OR' HASH IL+2 > OBJ STR SIZE M
348           'THEN' ERROR HANDLER PG('LITERAL'(E),"TOO%MANY%NAMES%INTRODUCED");
349        'END';
350     'END';
351
352  RETURN LL;
353  'END'FETCH ELEMENT PG;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT

TING OF :ZZZZZZZZZZZZN.STACKPG(8/)   PRODUCED ON 20JAN82 AT 19.03.51

PUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.59.57 USING I381

MENT   STACKPG(8/)

```
'PROCEDURE' STACK PG('VALUE' 'INTEGER' OPERAND TYPE Q);

'COMMENT'
PURPOSE
  TO PLANT CODE TO STACK THE OPERAND HELD IN "ELEMENT IG".
PARAMETERS
  OPERAND TYPE Q (THIS IS THE VALUE OF THE INSTRUCTION TO BE USED IN THE
                  STACK OPERATION OF THE MULE MACHINE)
GLOBAL VARIABLES
  ELEMENT IG (R)
  PROG MARK IG (RW)
MACROS
  INC M
  BYTE0 M
  BYTE1 M
  BYTE2 M
LOCAL VARIABLES
  LOOP END IL (LOOP CONTROL VARIABLE)
PROCEDURES CALLED
  ERROR HANDLER PG
MESSAGES PRODUCED
  3  NOR PROGRAM SPACE LEFT
METHOD
  1. THE STACK INSTRUCTION IS PUT INTO PROGRAM MEMORY
  2. THE OPERAND IS PUT INTO MEMORY
  3. MEMORY IS CHECKED FOR OVERFLOW
;

'BEGIN'
   'INTEGER' KL,LOOP END IL;
   PROGRAM DG[PROG MARK IG ] := OPERAND TYPE Q;

   'IF' OPERAND TYPE Q < 4
   'THEN'
   'BEGIN'
      PROGRAM DG[PROG MARK IG+1] := BYTE2M(ELEMENT IG);
      PROGRAM DG[PROG MARK IG+2] := BYTE1M(ELEMENT IG);
      PROGRAM DG[PROG MARK IG+3] := BYTE0M(ELEMENT IG);
      INC M(PROG MARK IG,4);
   'END'
   'ELSE'
   'BEGIN'

'COMMENT' STACKING A CHARACTER STRING LITERAL;

INC M(PROG MARK IG,1);
      LOOP END IL :=[ELEMENT IG]+ELEMENT IG; ("ELEMENT IG" POINTS
  AT THE STARTING LOCATION OF THE STRING AT WHICH THE CHARACTER
  STRING LENGTH IS STORED)

      'FOR' KL := ELEMENT IG 'STEP' 1 'UNTIL' LOOP END IL
      'DO'
      'BEGIN'
         PROGRAM DG[PROG MARK IG] := [KL];
         INC M(PROG MARK IG,1);
      'END';

      'IF' PROG MARK IG>PROG SIZE M
      'THEN' ERROR HANDLER PG('LITERAL'(E),"NO%PROGRAM%SPACE%LEFT");
'END';
   'END'STACK PG;
```

BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3

^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.FUNCTIONCALL(5/) PRODUCED ON 24AUG81 AT 18.54.58

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.06.10 USING I381

DOCUMENT   FUNCTIONCALL(5/)

```

'PROCEDURE' FUNCTION CALL PG;

'COMMENT'
PURPOSE
  DEALS WITH THE MULE FUNCTION CALL OPERATION
GLOBAL VARIABLES
  PROGRAM DG (W)
  PROG MARK IG (RW)
MACROS
  RAT FN CALL M
  INC M
  PROG SIZE M
PROCEDURES CALLED
  ERROR HANDLER PG
MESSAGES PRODUCED
  3  NO PROGRAM SPACE LEFT
METHOD
  JUST PUTS THE FUNCTION CALL INSTRUCTION INTO PROGRAM MEMORY
;

'BEGIN'
PTRACE M (TA PROC M,"FUNCCALL");
   PROGRAM DG[PROG MARK IG] := RAT FN CALL M;
   INC M(PROG MARK IG,1);

   'IF' PROG MARK IG > PROG SIZE M
   'THEN' ERROR HANDLER PG('LITERAL'(E),"NO%PROGRAM%SPACE%LEFT");
'END'FUNCTION CALL PG;
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.INFIXPG(13/) PRODUCED ON 17MAR82 AT 13.34.08

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.01.07 USING I381

DOCUMENT   INFIXPG(13/)

```
'RECURSIVE' INFIX PG;

'COMMENT'
PURPOSE
  DEALS WITH THE MULE INFIX STACKING OPERATION
GLOBAL VARIABLES
  ELEMENT IG (RW)
  ELEMENT TYPE IG (RW)
LOCAL VARIABLES
  ELE IL (STORE FOR ELEMENT IG)
  FINAL ELE IL, FINAL ELE TYPE IL (THE CLOSING OPERATOR, IE
                    THE CLOSE ROUND BRACKET IN THE CASE OF (A+B) -
                    THE THIRD OPERATOR IN THE CASE OF (A+B+.....) ETC)
MACROS
  ELE OPERAND M
  RAT AT M
  RAT CRD M
  RAT ST VAL M
  REPEAT M
  RAT ORD CRD M
PROCEDURES CALLED
  LEXIS PG (RECURSIVELY)
  FETCH ELEMENT PG
  FUNCTION CALL PG
  STACK PG
  ERROR HANDLER PG
MESSAGES PRODUCED
  1  SYNTAX ERROR - UNKNOWN ELEMENT ..
  7 THE OPERATOR IS NOT VALID
METHOD
  1. THE INFIX STACKING OPERATION IS OF THE FORM "(OPERAND1 OPERATOR OPERAND2)"
     WHERE BOTH OPERAND1 AND OPERAND2 CAN BE INFIX STACKING
     OPERATIONS THEMSELVES
  2. LEXISPG IS CALLED RECURSIVELLY TO SORT OUT OPERAND1. IT MAY CALL
     "INFIX PG" AGAIN.
  2A. THE OPERATOR MAY BE A FUNCTION CALL IN WHICH CASE IT IS NOT THE
     THE INFIX OPERATOR AND "LEXISPG" MUSRT BE CALLED AGAIN
     TO FIND THE OPERATOR - AND SO ON.
  3. THE OPERATOR MAY BE A SPECIAL SYMBOL,EG +,-,*,//,==£ OR A FUNCTION
     NAME. IT IS FETCHED, VIA "FETCH ELEMENT PG",AND CHECKED FOR VALIDITY.
  4. "LEXIS PG" IS CALLED AGAIN TO DEAL WITH OPERAND2.
;

'BEGIN'
   'INTEGER' ELE IL,FINAL ELE IL,FINAL ELE TYPE IL;

'COMMENT'
CALL "LEXIS PG" TO CAUSE STACKING OF THE FIRST OPERAND WHETGER IT IS
AN INFIX OPERATION OR NOT;

    LEXIS PG;

INFIX OPERATOR LL:
```

```
'COMMENT' FETCH THE OPERATOR;

   FETCH ELEMENT PG;

'COMMENT' CHECK FOR VALIDITY;


DIADIC AND RHS LL;
   'IF' ELEMENT TYPE IG <> ELE OPERAND M
   'OR' ELEMENT IG = RAT AT M
   'THEN'
   'BEGIN'
      ERROR HANDLER PG('LITERAL'(E),"THE%OPERATION%IS%NOT%VALID");
   'END'
   'ELSE'
   'BEGIN'

'COMMENT' CHECK TO SEE IF ITS A FUNCTION CALL OPERATOR AND IF SO
CALL THE FUNCTION CALL HANDLING PROCEDURE ("FUNCTION CALL PG") AND TRY
AGAIN;


      'IF' ELEMENT IG = RAT ORD CRD M
      'THEN'
      'BEGIN'
         FUNCTION CALL PG;
         REPEAT M(INFIX OPERATOR LL);
      'END'
'ELSE''IF' ELEMENT IG = RAT CRD M 'THEN' 'BEGIN'.
'COMMENT' THIS IS THE CASE OF A SINGLE OPERAND SURROUNDED BY BRACKETS,EG
(X) OR (X() ) IN WHICH CASE A RETURN IS PERFORMED ;
RETURN M;
'END';

'COMMENT' REMEMBER THE FETCHED ELEMENT; .....

      ELE IL := ELEMENT IG;

'COMMENT' STACK THE SECOND OPERAND VIA "LEXIS PG";

      LEXIS PG;

   INFIX END LL:

'COMMENT' FETCH NEXT ELEMENT - IT COULD BE THE CLOSING ROUND BRACKET
, A FUNCTION CALL OPERATOR OR ANOTHER OPERATOR ,EG A+B+C....
SO IF IT IS NOT THE PROGRAM IS IN ERROR;

      FETCHELEMENT PG;

'COMMENT' CHECK TO SEE IF ITS A FUNCTION CALL OPERATION AND IF SO
CALL THE FUNCTION CALL HANDLING PROCEDURE ("FUNCTION CALL PG") AND TRY
AGAIN FOR THE CLOSED ROUND BRACKET;


      'IF' ELEMENT IG = RAT ORD CRD M
      'THEN'
      'BEGIN'
         FUNCTION CALL PG;
         REPEAT M(INFIX END LL);
      'END';
FINAL ELE IL := ELEMENT IG;
FINAL ELE TYPE IL := ELEMENT TYPE IG;
```

```
        ELEMENT IG := ELE IL ;
        ELEMENT TYPE IG := ELE OPERAND M;
        STACK PG(RAT ST VAL M);
        FUNCTION CALL PG;
'IF' FINAL ELE IL<>RAT CRD M 'THEN'
'BEGIN'
'COMMENT' THIS IS THE CASE OF ANOTHER OPERATOR FOLLOWING, EG A+B+C*D;
ELEMENT IG := FINAL ELE IL;
ELEMENT TYPE IG := FINAL ELE TYPE IL;
REPEAT M(DIADIC AND RHS LL);
'END';
   'END';
RETURN LL;
'END'INFIX PG;
```

Â***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B***^

```
TING OF :ZZZZZZZZZZZN.ENDFUNCTIOND(8/)  PRODUCED ON 24AUG81 AT 18.57.08

PUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.53.01 USING I381

MENT    ENDFUNCTIOND(8/)


  0
  1
  2 'PROCEDURE' END FUNCTION DEF PG;
  3
  4 'COMMENT'
  5 PURPOSE
  6   TO MARK THE END OF A FUNCTION DEFINITION. WHAT HAS BEEN GENERATED
  7   TO DATE IS A BRANCH TO ZERO AT THE START OF THE FUNCTION, THE PROGRAM
  8   ADDRESS OF WHICH IS REMEMBERED IN "FUNCTION START IG". THE BRANCH TO
  9   ZERO MUST BE CHANGED TO A BRANCH TO THE CURRENT PROGRAM MEMORY LOCATION
 10   AND A STACK OF THE FUNCTION START LOCATION MADE.
 11 GLOBAL VARIABLES
 12   START HERE IG
 13   FUNCTION START DG (RW)
 14   FUNC NEST DEPTH IG (R/W)
 15   PROG MARK IG (RW)
 16   ELEMENT IG (W)
 17   ELEMENT TYPE IG (W)
 18 LOCAL VARIABLES
 19   REMEMBER PROG MARK IL (MEMORY FOR THE PROGRAM MEMORY MARK)
 20 MACROS
 21   ELE LITERAL M
 22   INC M
 23   RAT FN RET M
 24   RAT ST LIT M
 25 PROCEDURES CALLED
 26   ERROR HANDLER PG
 27   STACK PG
 28 MESSAGES PRODUCED
 29   6  FUNCTION END ENCOUNTERED WITHOUT A FUNCTION BEGIN
 30 METHOD
 31   1.  A CHECK IS MADE ON WHETHER A FUNCTION DEFINITION IS BEING
 32       PROCESSED . IF NOT THE PROGRAM IS IN ERROR.
 33   2.  THE BRANCH TO ZERO AT THE PROGRAM MEMORY LOCATION DENOTED BY THE STACK ITEM
 34       "FUNCTION START DG" IS CHANGED TO A BRANCH TO THE CURRENT MEMORY
 35       LOCATION.
 36   2A. A FUNCTION RETURN INSTRUCTION IS PLANTED BEFORE THE STACK INSTRUCTION
 37   3.  A STACK INSTRUCTION IS PLANTED USING THE FUNCTION START ADDRESS AS
 38       ITS OPERAND.
 39   4.  THE FUNCTION STACK IS "DOWN"ED TO DENOTE A REDUCTION IN THE FUNCTION NESTING LEVEL
 40 ;
 41
 42 'BEGIN'
 43    'INTEGER' REMEMBER PROG MARK IL;
 44
 45    'IF' FUNC NEST DEPTH IG < 0
 46    'THEN'
 47    'BEGIN'
 48       ERROR HANDLER PG('LITERAL'(E),"FUNCTION%END%ENCOUNTERED%WITHOUT%A%FUNCTION%BEGIN");
 49    'END'
 50    'ELSE'
 51    'BEGIN'
 52
 53 'COMMENT' PLANT A FUNCTION RETURN;
 54
 55       PROGRAM DG[PROG MARK IG] := RAT FN RET M;
 56       INC M(PROG MARK IG,1);
 57       REMEMBER PROG MARK IL := PROG MARK IG;
 58       PROG MARK IG := FUNCTION START DG[FUNC NEST DEPTH IG];
 59       ELEMENT IG := REMEMBER PROG MARK IG;
 60       ELEMENT TYPE IG := ELE LITERAL M;
 61       STACK PG(RAT ST LIT M);(PUT CURRENT PROGRAM MEMORY LOCATION INTO "BRANCH AROUND" CODE)
 62       FUNCTION START DG[FUNC NEST DEPTH IG] := PROG MARK IG+1;
 63 (UPDATE FUNCTION STACK TO POINT AT THE FUNCTION START NOW AND
 64    NOT THE BRANCH AROUND THE FUNCTION)
 65       PROG MARK IG := REMEMBER PROG MARK IL;(RESTORE THE PROGRAM MEMORY MARKER)
 66 'COMMENT' TELL THE INTERPRETER ("INTERP PG") WHERE TO START,
 67 IE AFTER THE FUNCTION DEFINITIONS;
 68 START HERE IG := PROG MARK IG;
 69       ELEMENT IG := FUNCTION START DG[FUNC NEST DEPTH IG] ;
 70       ELEMENT TYPE IG := ELE LITERAL M;
 71       STACK PG(RAT ST LIT M);(STACK THE FUNCTION START ADDRESS & THEN CLEAN UP)
 72       FUNC NEST DEPTH IG := FUNC NEST DEPTH - 1;
 73    'END'
 74 'END'END FUNCTION DEF PG;

*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*
```

STING OF :ZZZZZZZZZZZN.FUNCTIONDEFP(7/) PRODUCED ON 24AUG81 AT 18.58.26

TPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.52.19 USING I381

JMENT   FUNCTIONDEFP(7/)

```


'PROCEDURE' FUNCTION DEF PG;

'COMMENT'
PURPOSE
  THIS PROCEDURE IS CALLED WHEN THE "FUNCTION" LANGUAGE SYMBOL IS
  ENCOUNTERED. FUNCTIONS ARE ANONYMOUS IN MULE & HENCE THEIR
  VALUES NEED TO BE ASSIGNED TO AN INTRODUCED IDENTIFIER. THIS
  PROCEDURE CAUSES A BRANCH AROUND THE ACTUAL FUNCTION CODE TO BE
  GENERATED AND REMEMBERS THE START ADDRESS OF THE FUNCTION.
GLOBAL VARIABLES
  FUNCTION START DG (W)
  FUNC NEST DEPTH IG (R/W)
  PROG MARK IG (RW)
  PROGRAM DG (W)
  ELEMENT IG (W)
  ELEMENT TYPE IG (W)
LOCAL VARIABLES
  ELE IL (ELEMENT READ FROM THE MULE STREAM)
MACROS
  RAT END M
  ELE LITERAL M
  RAT BRANCH M
  RAT ST LIT M
  INC M
PROCEDURES CALLED
  ERROR HANDLER PG 'LITERAL'(E)
  FETCH ELEMENT PG
  STACK PG
MESSAGES PRODUCED
  5  NESTED FUNCTION DEFINITIONS ARE NOT PERMITTED
METHOD
  2. THE CURRENT PROGRAM MEMORY MARK IS USED TO REMEMBER THE START
     LOCATION OF THE FUNCTION.
  3. THE BRANCH AROUND THE FUNCTION CODE IS GENERATED.
;

'BEGIN'
  'INTEGER' ELE IL;

INC M(FUNC NEST DEPTH IG,1);
  'IF' FUNC NEST DEPTH IG > FUNC STC SIZE M
  'THEN'
  'BEGIN'
     ERROR HANDLER PG('LITERAL'(E),"NESTED%FUNCTION%DEPTH%EXCEEDED");
  'END'
  'ELSE'
  'BEGIN'
     FUNCTION START DG[FUNC NEST DEPTH IG] := PROG MARK IG;
(THIS IS TO REMEMBER WHERE THE FUNCTION STARTS IN MEMORY - EXCEPT THAT
   THE FUNCTION STARTS AFTER THE BRANCH WHICH IS ABOUT TO BE PLANTED)

'COMMENT' NOW PLANT A BRANCH AROUND THE FUNCTION CODE;

     ELEMENT IG := 0;
     ELEMENT TYPE IG := ELE LITERAL M;
     STACK PG(RAT ST LIT M);

'COMMENT' INITIALLY PUT IN A BRANCH TO ZERO (ABOVE STACK IS CONSTANT ZERO)
UNTIL THE FUNCTION END IS ENCOUNTERED. THE PRECEDING CODE PLANTED THE
"STACK CONSTANT 0 CODE" SO NOW WE PLANT THE BRANCH CODE ;

     PROGRAM DG[PROG MARK IG] := RAT BRANCH M;
     INC M(PROG MARK IG,1);
  'END'
'END'FUNCTION DEF PG;
```

*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZH.INTRODUCTION(6/)  PRODUCED ON 24AUG81 AT 19.01.28

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.54.12 USING I381

DOCUMENT    INTRODUCTION(6/)


 'PROCEDURE' INTRODUCTION PG;

 'COMMENT'
 PURPOSE
   THIS PROCEDURE DEALS WITH THE MULE "OBJECT INTRODUCTION" SYNTAX.
   IT USES THE PROCEDURE "FETCH ELEMENT PG" TO FETCH THE ELEMENTS OF
   MULE WHICH ALSO INTRODUCES THE OBJECTS. "FETCH ELEMENT PG" CAUSES
   THE OBJECT TO BE INTRODUCED WITH DEFAULT ATTRIBUTES. IF THE ATTRIBUTES

   ARE SPECIFIED EXPLICITLY IN THE INTRODUCTION THIS PROCEDURE
   CAUSES THE DATA TABLE RECORD TO BE OVERWRITTEN WITH THE SPECIFIED
   ATTRIBUTES.
 GLOBAL VARIABLES
   ELEMENT IG
   ELEMENT TYPE IG
   IN INTRO BG (W)
 LOCAL VARIABLES
   RAND IL  (OPERAND INDEX NUMBER)
 MACROS
   RAT COLON M
   ELE OPERAND M
   ELE LITERAL M
   RETURN M
   OBJ SIZE M
   RAT DOT M
   REPEAT M
   OBJ SIZE M
   TRUE M
   FALSE M
 PROCEDURES CALLED
   FETCH ELEMENT PG
   ERROR HANDLER PG
 MESSAGES PRODUCED
   4  SYNTAX ERROR - A NUMERIC CONSTANT MUST BE SPECIFIED AS AN ATTRIBUTE
      OF THE OBJECT INTRODUCTION
   1  SYNTAX ERROR - UNKNOWN ELEMENT ..
 METHOD
   1. AN ELEMENT OF MULE IS OBTAINED VIA "FETCH ELEMENT PG"
   2. IF THE ELEMENT IS AN OPERAND (IE A NAME) "FETCH ELEMENT PG" WILL
      HAVE INTRODUCED THE NAME ALREADY.
   3. THE NEXT ELEMENT IS FETCHED & CHECKED AGAINST THE ":" ELEMENT
      WHICH DENOTES THAT THE SIZE ATTRIBUTE IS BEING SPECIFIED
      IF THE ":" IS PRESENT THE SIZE ATTRIBUTE IS FETCHED
   4. IF ALL INTRODUCTIONS HAVE BEEN OBTAINED THE PROCEDURE RETURNS
 ;

 'BEGIN'
   'INTEGER' RAND IL;
   IN INTRO BG := TRUE M; (SET TO DENOTE THAT AN INTRODUCTION
```

```
54 CHECK ELEMENT LL;
55
56    'IF' ELEMENT TYPE IG= ELE OPERAND M
57    'THEN'
58    'BEGIN'
59 'IF' ELEMENT IG=RAT DOT M 'THEN' RETURN M;
60       RAND IL := ELEMENT IG; (REMEMBER DATA TABLE INDEX)
61       FETCH ELEMENT PG; (FETCH ANOTHER ELEMENT AND CHECK IT FOR A COLON)
62
63       'IF' ELEMENT IG = RAT COLON M
64       'THEN'
65       'BEGIN'
66          FETCH ELEMENT PG; (FETCH ANOTHER ELEMENT, EXPECT A NUMBER WHICH IS
67   THE SIZE ATTRIBUTE)
68
69          'IF' ELEMENT TYPE IG<>ELE LITERAL M
70          'THEN'
71          'BEGIN'
72             ERROR HANDLER PG('LITERAL'(E),
73 "SYNTAX%ERROR%-%A%NUMERIC%CONSTANT%MUST%BE%SPECIFIED%AS%AN%ATTRIBUTE%OF%THE%OBJECT%INTRODUCTION");
74             RETURN M;
75          'END'
76          'ELSE'
77          'BEGIN'
78             OBJ SIZE M[RAND IL]:= ELEMENT IG; (UPDATE SIZE ATTRIBUTE)
79          'END';
80       'END'
81       'ELSE' REPEAT M(CHECK ELEMENT LL);
82    'END'
83    'ELSE'
84    'BEGIN'
85
86       ERROR HANDLER PG('LITERAL'(E),"SYNTAX%ERROR%-%UNKNOWN%ELEMENT");
87       RETURN M;
88    'END';
89    REPEAT M(START LL);
90
91 RETURN LL;
92 IN INTRO BG := FALSE M ; (NO LONGER PROCESSING AN INTRODUCTION)
93 'END'INTRODUCTION PG;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZH.DESTACKPG(5/) PRODUCED ON 24AUG81 AT 19.02.17

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.59.17 USING I381

DOCUMENT    DESTACKPG(5/)

'RECURSIVE' DESTACK PG;

'COMMENT'
PURPOSE
  DEALS WITH THE DESTACKING OPERATION MULE SYNTAX
GLOBAL VARIABLES
  PROG MARK IG (RW)
  PROGRAM DG (W)
PROCEDURES CALLED
  LEXIS PG
  ERROR HANDLER PG
MACROS
  RAT DESTACK M
  PROG SIZE M
MESSAGES PRODUCED
  3 NO PROGRAM SPACE LEFT
METHOD
  1. CALLS "LEXIS PG" TO DEAL WITH THE STACKING OF THE RIGHT HAND
     OPERAND OF THE DESTACK
  2. PLANTS THE DESTACKING CODE INTO PROGRAM MEMORY
;

'BEGIN'

'COMMENT' CALL "LEXIS PG" TO DEAL WITH THE RIGHT HAND OPERAND;

   LEXIS PG;

'COMMENT' PLANT THE DESTACK OPERATION;

   PROGRAM DG[PROG MARK IG] := RAT DESTACK M;
   INC M(PROG MARK IG,1);

   'IF' PROG MARK IG > PROG SIZE M
   'THEN' ERROR HANDLER PG('LITERAL'(E),"NO%PROGRAM%SPACE%LEFT");
'END'DESTACK PG;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.POPVALUEPG(16/)   PRODUCED ON 26JAN82 AT 13.23.05

£OUTPUT BY LISTFILE IN ':TLFAK=10.RPSMITH' ON 16AUG82 AT 10.52.36 USING I381

DOCUMENT    POPVALUEPG(16/)


      0
      1
      2 'INTEGER' 'PROCEDURE' POP VALUE PG ('VALUE' 'INTEGER' SIZE DEFAULT Q);
      3
      4 'COMMENT'
      5 PURPOSE
      6   TO  POP THE STACK VALUE AND CONVERT IT INTO A VALUE SIGNIFICANT TO
      7   THE NUMBER OF BYTES SPECIFIED IN ITS PARAMETER (SUBJECT TO
      8   HOST CONSTRAINTS ALSO).
      9 GLOBAL VARIABLES
     10   STACK TYPE DG (R)
     11   STACK DG (R)
     12   STACK MARK IG (RW)
     13   OBJ TAB DG (R)
     14   MEMORY DG (R)
     15 LOCAL VARIABLES
     16   OPERAND IL (INDEX INTO THE DATA TABLE FOR THE OPERAND)
     17   SIZE IL (OPERAND MEMORY CELL SIZE)
     18   MAX SIZE IL (OPERAND SIZE IF < "SIZE DEFAULT Q" OR "SIZE DEFAULT Q" OTHERWISE)
     19   STEP START IL, STEP END IL, KL (LOOP CONTROL VARIABLES)
     20 MACROS
     21   LIT HELD M
     22   ADDR HELD M
     23   OBJ SIZE M (R)
     24 PARAMETERS
     25   SIZE DEFAULT Q (THE NUMBER OF MEMORY CELLS OF THE STACKED VALUE WHICH ARE
     26                   SIGNIFICANT)
     27 METHOD
     28   1. CHECK HE TYPE OF "VALUE" HELD ON THE STACK
     29   2. A CONSTANT IS MERELY RETURNED
     30   3. AN "ADDRESS" IS FETCHED FROM THE DATA TABLE RECORD & IS RETURNED
     31   4. A VALUE IS CONVERTED SO THAT ITS VALUE DOES NOT EXCEED THE DEFAULT
     32      SIZE OF THE PARAMETER & IF IT DOES THE VALUE IS TRUNCATED
     33 ;
     34
     35 'BEGIN'
     36    'INTEGER' OPERAND IL,SIZE IL,MAX SIZE IL,STEP START IL,STEP END IL,KL,ANSWER IL;
     37    PTRACE M (TA PROC M,"POPVALU");
     38
     39 'COMMENT' CHECK FOR THE CASE OF THE STACK VALUE BEING A LITERAL ;
     40
     41
     42    'IF' STACK TYPE DG[STACK MARK IG] = LIT HELD M
     43    'THEN'
     44    'BEGIN'
     45       SIZE IL := STACK DG[STACK MARK IG];
     46       STACK MARK IG := STACK MARK IG-SIZE IL/4=(
     47 'IF' 'INTEGER' (SIZE IL/4)*4=SIZE IL
     48       'THEN' 0
     49       'ELSE' 1);
     50       ANSWER IL := STACK DG[STACK MARKIG] ;(ONLY THE LEAST SIGNIFICANT 4 BYTES ARE WANTED)
     51    'END'
     52    'ELSE'
     53
```

```
55
56
57      'IF' STACK TYPE DG[STACK MARK IG ]=ADDR HELD M
58      'THEN' ANSWER IL := OBJ TAB DG[STACK DG[STACK MARK IG]+1]
59      'ELSE'
60      'BEGIN'
61
62  'COMMENT' CASE OF A VALUE BEING HELD;
63
64          OPERAND IL := STACK DG[STACK MARK IG];
65
66  'COMMENT' CHECK IF THE ADDRESS IS NEGATIVE & IF SO JUST RETURN THE
67  ADDRESS AS IT IS A BUILT IN ADDRESS WHICH HAS NO VALUE;
68
69
70          'IF' OBJ TAB DG[OPERAND IL+1]<0
71          'THEN' ANSWER IL := OBJ TAB DG[OPERAND IL+1]
72          'ELSE'
73          'BEGIN'
74             SIZE IL := OBJ SIZE M[OPERAND IL];
75             MAX SIZE IL :=
76
77             'IF' SIZE IL>SIZE DEFAULT Q
78             'THEN' SIZE DEFAULT Q
79             'ELSE' SIZE IL;
80
81  'COMMENT' RETURN THE LEAST SIGNIFICANT MEMORY CELLS (AS AN INTEGER) WHICH
82  MUST BE LESS THAN OR EQUAL TO "SIZE DEFAULT Q";
83
84             STEP END IL := OBJ TAB DG[OPERAND IL+1]+SIZE IL;
85             ANSWER IL := 0;
86             STEP START IL :=STEP END IL-MAX SIZE IL;
87
88             'FOR' KL :=STEP START IL,KL+1 'WHILE' KL<STEP END IL
89             'DO' ANSWER IL :=ANSWER IL *64 +MEMORY DG[KL];
90          'END';
91      'END';
92      STACK MARK IG := STACK MARK IG-1;
93
94      'IF' STACK MARK IG < -1
95      'THEN' ERROR HANDLER PG('LITERAL'(R),"STACK%UNDERFLOW");
96      'ANSWER' ANSWER IL;
97  'END'POP VALUE PG;
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.POPADDRESSPG(2/)  PRODUCED ON  6MAY81 AT 13.10.23

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.53.54 USING I381

DOCUMENT    POPADDRESSPG(2/)


'INTEGER' 'PROCEDURE' POP ADDRESS PG;

'COMMENT'
PURPOSE
  TO  POP THE STACK VALU & CONVERT IT INTO ADDRESS FORMAT
MACROS
  DEFAULT OBJ SIZE M
METHOD
  USES "POP VALUE PG" CONVERTING THE VALUE TO ADDRESS MODE BY CONSIDERING
  "DEFAULT OBJ SIZE M" NUMBER OF MEMORY CELLS TO BE SIGNIFICANT
;

'BEGIN'
   PTRACE M(TA PROC M,"POPADD");
   'ANSWER' POP VALUE PG(DEFAULT OBJ SIZE M);
'END'POP ADDRESS PG;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.POPINTEGERPG(2/) PRODUCED ON 6MAY81 AT 13.08.31

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.53.18 USING I381

DOCUMENT   POPINTEGERPG(2/)

```

'INTEGER' 'PROCEDURE' POP INTEGER PG;

'COMMENT'
PURPOSE
  TO POP THE STACK VALUE & CONVERT IT TO INTERNAL ARITHMETIC (INTEGER)
  FORMAT
METHOD
  USES "POP VALUE PG" CONVERTING THE VALUE TO INTERNAL ARITHMETIC
  (INTEGER) FORMAT BY CONSIDERING 4 MEMORY CELLS AS SIGNIFICANT
;

'BEGIN'
   PTRACE M(TA PROC M,"POPINT");
   'ANSWER' POP VALUE PG(4);
'END'POP INTEGER PG;
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.BUILTINPG(47/)   PRODUCED ON 4MAR82 AT 19.31.29

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.04.27 USING I381

DOCUMENT   BUILTINPG(47/)

```

'PROCEDURE' BUILT IN PG('VALUE' 'INTEGER' BUILT IN ROUTINE SELECTOR Q);

'COMMENT'
PURPOSE
  CALLED FROM "INTERPRET PG" TO DEAL WITH THE NON-MULE BUILT
  IN ROUTINES WHICH CAN BE CALLED BY THE MULE USER
GLOBAL VARIABLES
  MEMORY DG (RW)
  OBJ TAB DG (W)
  STACK MARK IG (RW)
  STACK DG (RW)
  STACK TYPE DG (RW)
  RUN TIME PROG MARK IG (RW)
  FUNC STC MARK IG (RW)
PROCEDURES CALLED
  POP ADDRESS PG
  POP INTEGER PG
MACROS
  OBJ SIZE M
  I CH M
  ONL M
  OCH M
  EXIT M
  UP STACK M
  RETURN M
  LIT HELD M
LOCAL VARIABLES
  BUILT INS SL (THE MAIN SELECTION SWITCH)
  SIZE IL (LOCAL OBJECT SIZE)
  START LOOP IL, KL (LOOP COUNTERS)
  DIADICS SL (DIADIC OPERATORS SELECTION SWITCH)
  BYTE STACK DL (A BYTE ARRAY OVERLAYED ON THE ACTUAL RUN TIME STACK
    TO ALLOW ACCESS TO THE STACK BYTE BY BYTE)
  BYTE STACK MARK IL (STACK MARKER FOR BYTE STACK)
  ADDRESS IL (AN OPERANDS ADDRESS IN DATA MEMORY)
  CHARACTER STRING DL (CHARACTER STRING STORAGE)
  CHAR STR POINTER DL (2 WORD POINTER FOR RMCS CORAL STRING)
PARAMETERS
  BUILT IN ROUTINE SELECT Q (INDEX INTO MAIN SELECTION SWITCH)
METHOD
  THE MAIN SELECTION SWITCH STEERS CONTROL TH THE CODE TO DEAL WITH THE
  BUILT IN ROUTINE CALL
;

'BEGIN'
  'INTEGER' SIZE IL,START LOOP IL,KL,RH IL,LH IL,RESULT IL,BYTE STACK MARK IL,ADDRESS IL;
  'BYTE''ARRAY' CHARACTER STRING DL[0:63];
  'INTEGER' 'ARRAY' CHAR STR POINTER DL[0:1];
  'OVERLAY' STACK DG[0] 'WITH' 'BYTE''ARRAY' BYTE STACK DL[0:3];
  'SWITCH' BUILT INS SL :=MAKE BUILT IN LL,
  IN CHAR LL,
```

```
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    DIADICS LL,
    IF LL,
    GOTO LL,
    POP LL,
    STACK NAME LL,
    PRINT INT LL,
    COMPILE LL,
    SUBSTR LL,
    VALUE LL;
    PTRACE M(TA PROC M,"BUILTIN");
    'GOTO' BUILTINS SL[BUILT IN ROUTINE SELECTOR Q];

'COMMENT'
1: MAKEBUILTIN
PURPOSE
  TO ALTER THE DATA TABLE RECORD TO MAKE A KNOWN FUNCTION NAME INTO A BUILT
  IN FUNCTION NAME
PARAMETERS
  1. NAME OF THE BUILT IN FUNCTION, EG "NAME"
  2. A CONSTANT WHICH DENOTES THE BUILT IN FUNCTIONS NUMBER
;


MAKE BUILT IN LL:
    PTRACE M(TA PROC M,"IBMAKE BUILT IN ");
    RH IL := POP INTEGER PG;
    OBJ TAB DG[STACK DG[STACK MARK IG]+1] := -( RH IL );
    POP ADDRESS PG;(REMOVE THE ITEM FROM THE STACK)
    EXIT M;

'COMMENT'
2: INCHAR
PURPOSE
  TO READ A NUMBER OF CHARACTERS INTO A DATA AREA PASSED AS A PARAMETER.
  THE NUMBER OF CHARACTERS READ CORRESPONDS TO THE SIZE, IN MEMORY CELLS,
  OF THE DATA AREA.
PARAMETERS
  1. THE ADDRESS OF THE OBJECT WHICH IS TO RECEIVE THE CHARACTERS FROM A
     FILE, EG @BUFFER
;


IN CHAR LL:
    PTRACE M(TA PROC M,"IBIN CHAR ");
    SIZE IL := OBJ SIZE M[STACK DG[STACK MARK IG]];
    START LOOP IL := POP ADDRESS PG;

    'FOR' KL := START LOOP IL 'STEP' 1 'UNTIL' START LOOP IL+SIZE IL-1
    'DO' MEMORY DG[KL] := I CH M;
    EXIT M;

```

```
121    TO OUTPUT A NUMBER OF CHARACTERS FROM A DATA AREA TO A FILE. THE NUMBER
122    OF CHARACTERS OUTPUT IS GIVEN BY THE LENGTH OF THE DATA AREA.
123  PARAMETERS
124    1. THE ADDRESS OF THE BUFFER FROM WHICH THE CHARACTERS WILL BE OBTAINED
125       FOR OUTPUT, EG @BUFFER
126  ;
127
128
129  OUT CHAR LL:
130     PTRACE M(TA PROC M,"IHOUT CHAR ");
131     SIZE IL := OBJ SIZE M[STACK DG[STACK MARK IG]];
132     START LOOP IL := POP ADDRESS PG ;
133
134     'FOR' KL := START LOOP IL 'STEP' 1 'UNTIL' START LOOP IL+SIZE IL-1
135     'DO'
136     'BEGIN'
137
138        'IF' MEMORY DG[KL]=63
139        'THEN' ONL M
140        'ELSE' O CH M(MEMORY DG[KL]);
141     'END';
142     EXIT M;
143
144  DIADICS LL:
145     PTRACE M(TA PROC M,"IRDIADICS ");
146
147  'COMMENT'
148  DIADIC OPERATORS
149  4: +
150  5: -
151  6: *
152  7: /
153  8: ^
154  9: <
155  10: >
156  11: <=
157  12: >=
158  13: =
159  14: <>
160  15: AND
161  16: OR
162  17: UNALLOCATED
163  ;
164
165     RH IL := POP INTEGER PG;
166     LH IL := POP INTEGER PG;
167     'BEGIN'
168        'SWITCH' DIADICS SL := PLUS LL,
169        MINUS LL,
170        MULTIPLY LL,
171        DIVIDE LL,
172        EXPONENT LL,
173        LSS LL,
174        GRT LL,
175        LSE LL,
176        GRE LL,
177        EQU LL,
178        NEQ LL,
179        AND LL,
180        OR LL;
181        'GOTO' DIADICS SL[BUILT IN ROUTINE SELECTOR Q-3];
182
183     PLUS LL:
```

```
187
188   MINUS LL:
189      PTRACE M(TA PROC M,"IB   MINUS ");
190      RESULT IL := LH IL-RH IL;
191      EXIT M;
192
193   MULTIPLY LL:
194      PTRACE M(TA PROC M,"IB   MULTIPLY ");
195      RESULT IL := LH IL*RH IL;
196      EXIT M;
197
198   DIVIDE LL:
199      PTRACE M(TA PROC M,"IB   DIVIDE ");
200      RESULT IL := LH IL/RHIL;
201      EXIT M;
202
203   EXPONENT LL:
204      PTRACE M(TA PROC M,"IB   EXPONENT ");
205      EXIT M;
206
207   LSS LL:
208      PTRACE M(TA PROC M,"IB   LSS ");
209      RESULT IL :=
210
211      'IF' LH IL<RH IL
212      'THEN' 1
213      'ELSE' 0;
214      EXIT M;
215
216   GRT LL:
217      PTRACE M(TA PROC M,"IB   GRT ");
218      RESULT IL :=
219
220      'IF' LH IL>RH IL
221      'THEN' 1
222      'ELSE' 0;
223      EXIT M;
224
225   LSE LL:
226      PTRACE M(TA PROC M,"IB   LSE ");
227      RESULT IL :=
228
229      'IF' LH IL <= RH IL
230      'THEN' 1
231      'ELSE' 0;
232      EXIT M;
233
234   GRE LL:
235      PTRACE M(TA PROC M,"IB   GRE ");
236      RESULT IL :=
237
238      'IF' LH IL>=RH IL
239      'THEN' 1
240      'ELSE' 0;
241      EXIT M;
242
243   EQU LL:
244      PTRACE M(TA PROC M,"IB   EQU ");
245      RESULT IL :=
246
247      'IF' LH IL=RH IL
248      'THEN' 1
249      'ELSE' 0;
```

```
252    NEQ LL:
253       PTRACE M(TA PROC M,"IB   NEQ ");
254       RESULT IL :=
255
256       'IF' LH IL<>RH IL
257       'THEN' 1
258       'ELSE' 0;
259       EXIT M;
260
261    AND LL:
262       PTRACE M(TA PROC M,"IB   AND ");
263       RESULT IL := LH IL 'MASK' RH IL;
264       EXIT M;
265
266    OR LL:
267       PTRACE M(TA PROC M,"IB   OR ");
268       RESULT IL := LH IL 'UNION' RH IL;
269       EXIT M;
270
271    EXIT LL:
272       PTRACE M(TA PROC M,"IB   EXIT ");
273       UPSTACK M;
274       STACK DG[STACK MARK IG] := RESULT IL;
275       UPSTACK M;
276       STACK TYPE DG[STACK MARK IG] := LIT HELD M;
277       STACK DG[STACK MARK IG] := 4; (THIS IS THE MEMORY CELL SIZE OF THE STACK
278    ITEM)
279     'END';
280     EXIT M;
281
282 'COMMENT'
283
284 18: IF
285 PURPOSE
286   PROVIDES THE "IF ... THEN" FACILITY OF A CONVENTIONAL
287   HIGH LEVEL PROGRAMMING LANGUAGE
288 PARAMETERS
289   1. THE RESULT OF A "BOOLEAN" EXPRESSION
290   2. A FUNCTION WHICH WILL BE APPLIED IF THE FIRST PARAMETER EVALUATES
291      TO NON-ZERO
292   2. THE RESULT OF A "BOOLEAN" EXPRESSION
293 ;
294
295
296 IF LL:
297    PTRACE M(TA PROC M,"IBIF ");
298    ADDRESS IL := POP ADDRESS PG ;
299    RESULT IL := POP INTEGER PG;
300
301    'IF' RESULT IL<>0
302    'THEN'
303    'BEGIN'
304       RUN TIME PROG MARK IG := ADDRESS IL-1;
305       RETURN M;
306    'END';
307
308    EXIT M;
309
310 'COMMENT'
311 19: GOTO
312 PURPOSE
313   NOT INTENDED FOR USER CONSUMPTION. ALLOWS THE DIRECT BRANCH TO BE TAKEN
314 PARAMETERS
315   1. A CONSTANT WHICH DENOTES A RELATIVE LOCATION IN THE PROGRAM MEMORY
```

```
318
319
320 GOTO LL:
321    PTRACE M(TA PROC M,"IBGOTO ");
322    RUN TIME PROG MARK IG := RUN TIME PROG MARK IG+POP INTEGER PG-1;
323    EXIT M;
324
325 'COMMENT'
326 20: POP
327 PURPOSE
328   TO REMOVE A RETURN ADDRESS FROM THE FUNCTION STACK TO ALLOW A FAST RETURN
329   TO BE PROVIDED
330 NO PARAMETERS
331 ;
332
333
334 POP LL:
335    PTRACE M(TA PROC M,"IBPOP");
336    FUNC STC MARK IG := FUNC STC MARK IG-1;
337    EXIT M;
338
339 'COMMENT'
340 21: STACKNAME
341 PURPOSE
342   TO PUT THE STRING REPRESENTING A NAME OF A VARIABLE ON THE STACK GIVEN
343   ITS OBJECT TABLE INDEX
344 PARAMETERS
345   1. THE NAME OF THE VARIABLE
346 ;
347
348
349 STACK NAME LL:
350    PTRACE M(TA PROC M,"IBSTNAM");
351    START LOOP IL := OBJ STR MARK M[STACK DG[STACK MARK IG]];(THE OBJECT STRING TABLE OFFSET)
352    POP ADDRESS PG; (REMOVE AN ITEM FROM THE STACK)
353    UPSTACK M;
354    BYTE STACK MARK IL := STACK MARK IG*4;
355    SIZE IL := OBJECT STRING DG[START LOOP IL];
356
357    'FOR' KL := START LOOP IL+1 'STEP' 1 'UNTIL' START LOOP IL+SIZE IL
358    'DO'
359    'BEGIN'
360       BYTE STACK DL[BYTE STACK MARK IL] := OBJECT STRING DG[KL];
361       INC M(BYTE STACK MARK IL,1);
362    'END';
363    STACK MARK IG := BYTE STACK MARK IL/4+(
364 'IF' 'INTEGER' (SIZE IL/4)*4=SIZE IL
365    'THEN' 0
366    'ELSE' 1);
367    STACK DG[STACK MARK IG] := SIZE IL ;
368    STACK TYPE DG[STACK MARK IG] := LIT HELD M;
369    EXIT M;
370
371 'COMMENT'
372 22: OUTINT
373 PURPOSE
374   TO DISPLAY THE VALUE OF AN OBJECT IN INTEGER FORMAT
375 PARAMETERS
376   1. THE NAME OF THE VARIABLE
377 ;
378
379
380 PRINT INT LL:
381    PTRACE M(TA PROC M,"IBOUTINT");
382    OINT M(POP INTEGER PG);
```

```
384     EXIT M;
385
386 'COMMENT'
387 23: COMPILE
388 PURPOSE
389   TO CAUSE THE TRANSLATION OF A MULE FUNCTION WHICH IS STORED IN A FILE
390 PARAMETERS
391   1. A CHARACTER STRING LITERAL WHICH REPRESENTS THE FILE NAME
392 ;
393
394
395 COMPILE LL:
396     PTRACE M(TA PROC M,"IHCOMPILE");
397
398 'COMMENT' CREATE A CHARACTER STRING IN RMCS CORAL FORM - IMPLEMENTATION
399 DEPENDENT;
400
401     KL := 0;
402     START LOOP IL := STACK MARK IG-STACK DG[STACK MARK IG]/4 -(
403 'IF' 'INTEGER' (STACK DG[STACK MARK IG]/4)*4=STACK DG[STACK MARK IG]
404     'THEN' 0
405     'ELSE' 1);
406
407 START LOOP IL := START LOOP IL*4; (BYTE STACK POSITION)
408     'FOR' START LOOP IL := START LOOP IL 'STEP' 1 'UNTIL' START LOOP IL+STACK DG[STACK MARK IG]-1
409     'DO'
410     'BEGIN'
411        CHARACTER STRING DL[KL] :=BYTE STACK DL[START LOOP IL];
412        INC M(KL,1);
413     'END';
414     CHAR STR POINTER DL[0] :='LOCATION' (CHARACTER STRING DL[0]);
415     CHAR STR POINTER DL[1] := STACK DG[STACK MARK IG];
416     POP INTEGER PG; (REMOVE ITEM FROM THE STACK)
417
418 'COMMENT' NOW SEND A MESSAGE TO GEORGE 3 TO ALLOW HIM
419 TO ASSIGN THE FILE;
420
421     MULE MESSAGE M (IN,'LOCATION' (CHAR STR POINTER DL[0]));
422     ISEL M(IF COMPILE M);
423     EXIT M;
424
425 'COMMENT'
426 24: SUBSTR
427 PURPOSE
428   TO PROVIDE THE VECTOR INDEXING ABILITY. SUPPOSE A MULTI CELLED OBJECT WAS
429   INTRODUCED AS
430      USE VECTOR : 30 .
431   THE VECTOR CAN BE CONSIDERED AS 30 SINGLE CELLS, 15 DUAL CELLED OBJECTS.. ETC
432   THIS FUNCTION PROVIDES THE VECTOR SUBSTRING CAPABILITY & RETURNS TO THE
433   STACK THE LOCATION OF AN ELEMENT OF THE VARIABLE AS SPECIFIED
434   BY THE PARAMETERS.
435 PARAMETERS
436   1. THE LOCATION OF THE VECTOR, EG @VECTOR
437   2. THE START CELL NUMBER,FIRST CELL IS ZERO
438   3. THE SIZE OF THE SUBSTRING IN MEMORY CELLS
439 ;
440
441
442 SUBSTR LL:
443     PTRACE M(TA PROC M,"IBSUBSTR");
444     RH IL :=POP INTEGER PG; (SIZE OF SUBSTRING)
445     LH IL := POP INTEGER PG ; (START CELL NUMBER)
446     ADDRESS IL := POP ADDRESS PG; (START ADDRESS OF THE VECTOR)
447
```

```
450
451     'IF' STACK TYPE DG[STACK MARK IG+1]<>ADDR HELD M
452     'THEN'
453     'BEGIN'
454
455 'COMMENT' THIS IS AN EVALUATED ADDRESS CONSTANT WHICH IS OF
456 THE FOLLOWING FORMAT
457     BOTTOM 18 BITS - THE MEMORY ADDRESS
458     TOP 6 BITS     - THE OBJECTS SIZE;
459
460         SIZE IL := 'BITS'[6,18]ADDRESS IL;
461         ADDRESS IL := 'BITS'[18,0]ADDRESS IL;
462
463         'IF' SIZE IL = 0
464         'THEN' ERRORHANDLER PG('LITERAL'(R),"THE%%VALUE
465 %PASSED%TO%'SUBSTR'%IS%NOT%AN%ADDRESS%VALUE");
466     'END'
467     'ELSE' SIZE IL := OBJ SIZE M[STACK DG[STACK MARK IG+1]];
468
469     'IF' RH IL+LH IL>SIZE IL
470     'THEN' ERROR HANDLER PG('LITERAL'(R),"SUBSTRING%IS%OUTSIDE%THE%VECTOR%RANGE");
471     UPSTACK M;
472     STACK DG[STACK MARK IG] :=ADDRESS IL+LH IL+RH IL*262144;
473 (VECTOR ADDRESS WITH THE SIZE IN THE TOP 6 BITS)
474     UPSTACK M;
475     STACK TYPE DG[STACK MARK IG] := LIT HELD M;
476     STACK DG[STACK MARK IG] :=4; (LITERAL HELD WITH A LITERAL SIZE OF 4 - IMPLEMENTATION DEPENDENT)
477     EXIT M;
478
479 'COMMENT'
480 25: VALUE
481 PURPOSE
482   TO PROVIDE A DEREFERENCING FACILITY.
483     CAUSES THE VALUE OF THE OBJECT REFERENCED TO BE PLACED UPON THE STACK
484 PARAMETERS
485   1. THE ADDRESS TO BE DEREFERENCED
486 ;
487
488
489 VALUE LL:
490 PTRACE M(TA PROC M,"IR VALUE");
491
492     'IF' STACK TYPE DG[STACK MARK IG]<> ADDR HELD M
493     'THEN'
494     'BEGIN'
495
496 'COMMENT' CASE OF AN ADDRESS VALUE WHICH IS STORED AS
497     TOP 6 BITS - REFERENCED OBJECT SIZE
498     BOTTOM 18 BITS - REFERENCED OBJECT ADDRESS
499 ;
500
501         ADDRESS IL := POP ADDRESS PG;
502         SIZE IL := 'BITS'[6,18]ADDRESS IL;
503         ADDRESS IL := 'BITS'[18,0]ADDRESS IL;
504     'END'
505     'ELSE'
506     'BEGIN'
507
508 'COMMENT' CASE OF AN ADDRESS CONSTANT ;
509
510         SIZE IL := OBJ SIZE M[STACK DG[STACK MARK IG]];
511         ADDRESS IL := POP ADDRESS PG;
```

```
    'FOR' KL := ADDRESS IL 'STEP' 1 'UNTIL' ADDRESS IL+SIZE IL-1
    'DO'
    'BEGIN'
       BYTE STACK DL[BYTE STACK MARK IL] := MEMORY DG[KL];
       INC M(BYTE STACK MARK IL,1);
    'END';
    STACK MARK IG :=STACKMARKIG+'INTEGER'(SIZE IL/4)+(

'IF' 'INTEGER' (SIZE IL/4)*4=SIZE IL
    'THEN' 0
    'ELSE' 1);
    STACK TYPE DG[STACK MARK IG] := LIT HELD M;
    STACK DG[STACK MARK IG] := SIZE IL;
    EXIT M;

EXIT LL:
    PTRACE M(TA PROC M,"IBEXIT ");

'COMMENT'
CAUSE A "RETURN" OPERATION TO THE MULE ABSTRACT MACHINE BY DOWNING
THE FUNCTION STACK;

    FUNC STC MARK IG := FUNC STC MARK IG-1;

'COMMENT' & THEN TRANSFER CONTROL TO THE NEXT INSTRUCTION BY SIMPLY
RETURNING AS "INTERPRET  PG" DEALS WITH THE INCREMENTING OF
THE PROGRAM COUNTER;


RETURN LL:
    PTRACE M(TA PROC M,"IBRETURN ");
'END'BUILT IN PG;
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.INTERPRETPG(48/)  PRODUCED ON 26JAN82 AT 13.47.32

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.55.22 USING I381

DOCUMENT    INTERPRETPG(48/)


      0
      1
      2 'PROCEDURE' INTERPRET PG;
      3
      4 'COMMENT'
      5 PURPOSE
      6   TO INTERPRET THE MULE OPERATIONS & PROVIDE MULE EXECUTION FACILITIES
      7 GLOBAL VARIABLES
      8   RUN TIME PROG MARK IG (RW)
      9   START HERE IG (R)
     10   PROGRAM DG (R)
     11   STACK TYPE IG (W)
     12   STACK MARK IG (RW)
     13   STACK DG (RW)
     14   MEM MARK IG (RW)
     15   OBJ TAB DG (R)
     16   FUNC MEMORY DG (RW)
     17   FUNC STC MARK G (RW)
     18   FUNC STACK DG (RW)
     19   FUNC NAME DG (RW)
     20 LOCAL VARIABLES
     21   OPERATOR IL (THE OPERATOR FETCHED)
     22   OP MOD IL (THE OPERATION MODIFIER)
     23   OP CODE IL (THE OPERATION CODE)
     24   INTERP ERROR IL (INTERNAL ERROR CODE)
     25   SIZE IL (OPERAND SIZE )
     26   OBJ ADD IL (OPERAND ADDRESS)
     27   CASE 1 SL/CASE1 X SL/CASE2 X SL (SELECTION SWITCHES)
     28   MAX RT PROG MARK IL (THE MAXIMUM VALUE OF "RUN TIME PROG MARK IG"
     29     DURING THIS INTERPRETATION)
     30 MACROS
     31   VAL HELD M
     32   ADDR HELD M
     33   LIT HELD M
     34   OBJ SIZE M (RW)
     35   MEM SIZE M
     36   FUNC STC SIZE M
     37   UP STACK M
     38 PROCEDURES CALLED
     39   POP ADDRESS PG
     40   BUILT IN PG
     41 METHOD
     42   1. INTERPRETATION IS BY MEANS OF READING THE PROGRAM WHICH IS IN PROGRAM
     43      MEMORY STARTING AT "START HERE IG" UNTIL EITHER A "HALT" INSTRUCTION
     44      IS MET, OR A RUN TIME ERROR OCCURS. BOTH OF THESE CAUSE A RETURN TO
     45      THE CALLING PROCEDURE "LEXIS PG".
     46   2. THE OPERATION IS DECODED INTO THE OPERATION AND OPERATION
     47      MODIFIER FIELDS.
     48   3. CONTROL IS SWITCHED ON THE OPERATION FIELD TO CASE=1 (NOTES 4 TO 10 HERE)
     49      AND CASE=2 (NOTES 11 TO 14 HERE)
     50   4. OPERATION FIELD ZERO OPERATIONS HAVE AN OPERAND FOLLOWING THE OPERATOR.
     51      THIS IS FETCHED INTO "OPERAND IG".
```

```
55        ONTO THE MAIN STACK AND THE STACK MARKED FOR "VALUE OPERAND HELD".
56        "OPERAND IG" IS AN INDEX INTO THE OPJECT TABLE WHERE THE ADDRESS OF THE
57        OBJECT CAN BE OBTAINED, AND HENCE ITS VALUE.
58     8. CASE-1.3: A  "STACK ADDRESS" OPERATION CAUSES "ADDRESS HELD" AND
59        "OPERAND IG" TO BE COPIED TO THE STACK.
60     9. CASE-1.4: A "STACK LITERAL" OPERATION CAUSES THE OPERAND TO BE COPIED
61        TO THE STACK AND THE STACK MARKED AS "LITERAL HELD".
62    10. CASE-1.5: A "DECLARE OPERATION" CAUSES SPACE TO BE ALLOCATED
63        IN MEMORY FOR INTRODUCED OBJECTS & THEIR ADDRESSES TO BE PLACED IN THE
64        OBJECT TABLE.
65    11. CASE 2.1: A "FUNCTION CALL" OPERATION CAUSES THE FUNCTION
66        RETURN ADDRESS TO BE PUT ONTO THE FUNCTION STACK ("FUNC STACK DG")
67        TOGETHER WITH SOME DEBUG INFORMATION AND CAUSES CONTROL TO BE TRANSFERRED
68        TO THE INVOKED FUNCTION BY ALTERING THE RUN TIME PROGRAM MARKER
69        (RUN TIME PROG MARK IG)
70    12. CASE-2.1: A CALL OF A BUILT IN FUNCTION (SEE 11 ABOVE) IS DETECTED
71        BY THE STACKED VALUE BEING NEGATIVE. THIS PROCEDURE HANDLES BUILT
72        IN FUNCTIONS BY CALLING "BUILT IN PG"
73    13. CASE-2.2: A "DIRECT BRANCH OPERATION IS DEALT WITH BY ALTERING THE
74        PROGRAM MEMORY MARKER.
75    14. CASE-2.3: A "FUNCTION RETURN" OPERATION CAUSES THE FUNCTION STACK
76        TO BE DOWNED & A FUNCTION RETURN PERFORMED BY ALTERING THE PROGRAM
77        MEMORY MARKER.
78 ;
79
80 'BEGIN'
81    'INTEGER' OPERATOR IL,OP CODE IL,OP MOD IL,INTERP ERROR IL,SIZE IL,OBJ ADD IL,OPERAND IL,
82    MAX RT PROG MARK IL,LH SIZE IL,RH SIZE IL,KL,BYTE STACK MARK IL;
83    'OVERLAY' STACK DG[0] 'WITH' 'BYTE''ARRAY' BYTE STACK DL[0:1];
84    'SWITCH' CASE 1 SL :=CASE 1 LL,
85    CASE 2 LL;
86    'SWITCH' CASE 1 X SL :=CASE 11 LL,
87    CASE 12 LL,
88    CASE 13 LL,
89    CASE 14 LL,
90    CASE 15 LL,
91    CASE 16 LL;
92    'SWITCH' CASE 2 X SL :=EXIT LL,
93    CASE2 FN CALL LL,
94    CASE2 DIRECT BRANCH LL,
95    CASE2 FN RETURN LL,
96    CASE 2 DESTACK LL;
97    PTRACE M(TA PROC M,"INTERP");
98    RUN TIME PROG MARK IG := START HERE IG;
99    MAX RT PROG MARK IL := 0;
100
101 START LL:
102
103    'IF' MAX RT PROG MARK IL < RUN TIME PROG MARK IG
104    'THEN' MAX RT PROG MARK IL := RUN TIME PROG MARK IG;
105    OPERATOR IL := PROGRAM DG[RUN TIME PROG MARK IG];(GET THE OPERATOR & DECODE IT)
106    OP CODE IL := 'BITS'[2,4]OPERATOR IL; (THE OPERATION CODE)
107    OP MOD IL := 'BITS'[4,0]OPERATOR IL;(THE OPERATION MODIFIER)
108
109    'IF' OP CODE IL > 1
110    'THEN'
111    'BEGIN'
112       ERROR HANDLER PG('LITERAL'(R),"OPERATION%UNKNOWN");
113    'END'
114    'ELSE' 'GOTO' CASE 1 SL[OP CODE IL+1];
115
116 CASE 1 LL:
117
118 'COMMENT' OPERAND
```

```
121 'COMMENT' FETCH THE OPERAND;
122
123
124    'IF' OP MOD IL <> RAT ST CHAR LIT M
125    'THEN'
126    'BEGIN'
127       OPERAND IL := PROGRAM DG[RUN TIME PROG MARK IG+1]*64*64+PROGRAM DG[RUN TIME PROGMARKIG+2]
128       *64+PROGRAM DG[RUN TIME PROG MARK IG+3];
129       INC M(RUN TIME PROG MARK IG,3);
130    'END';
131
132    'IF' OP MOD IL>5
133    'THEN'
134    'BEGIN'
135       ERROR HANDLER PG('LITERAL'(R),"OPERATIONXMODIFIERX(O)XUNKNOWN");
136    'END'
137    'ELSE' 'GOTO' CASE 1 X SL[OP MOD IL+1];
138
139 CASE 11 LL:
140    PTRACE M(TA PROC M,"IHALT");
141
142 'COMMENT'OPERATION CODE IS ZERO - OPERATION MODIFIER IS HALT;
143
144
145 'COMMENT' REMOVE THE PROGRAM WHICH HAS BEEN STORED
146 INTERACTIVELY AS IT WANTS EXECUTING ONCE ONLY;
147
148
149    'FOR' OBJ ADD IL := START HERE IG 'STEP' 1 'UNTIL' MAX RT PROG MARK IL
150    'DO' PROGRAM DG[OBJ ADD IL] := 0;
151    PROG MARK IG := START HERE IG;
152    RETURN M;
153
154 CASE 12 LL:
155    PTRACE M(TA PROC M,"ISTCVAL");
156
157 'COMMENT' OPERATION CODE IS ZERO - OPERATION MODIFIER IS "STACK VALUE";
158
159    UP STACK M;
160
161 'COMMENT' MARK THE STACK AS "VALUE" HELD;
162
163    STACK TYPE DG[STACK MARK IG] := VAL HELD M;
164
165 'COMMENT' PUT DATA TABLE INDEX ON THE STACK;
166
167    STACK DG[STACK MARK IG] := OPERAND IL ;
168    EXIT M;
169
170 CASE 13 LL:
171    PTRACE M(TA PROC M,"ISTCADD");
172
173 'COMMENT' OPERATION CODE IS ZERO - OPERATION MODIFIER IS "STACK ADDRESS";
174
175    UP STACK M;
176
177 'COMMENT' MARK THE STACK AS "ADDRESS HELD";
178
179    STACK TYPE DG[STACK MARK IG] := ADDR HELD M;
180
181 'COMMENT' PUT DATA TABLE INDEX ONTO THE STACK;
182
    STACK DG[STACK MARK IG] := OPERAND IL;
```

```
187     PTRACE M(TA PROC M,"ISTCLIT");
188
189 'COMMENT' OPERATION CODE IS ZERO - OPERATION MODIFIER IS "STACK LITERAL";
190
191     UPSTACK M;
192
193 'COMMENT' PUT THE LITERAL ON THE STACK;
194
195     STACK DG[STACK MARK IG] := OPERAND IL;
196     UP STACK M;
197
198 'COMMENT' MARK THE STACK AS "LITERAL HELD";
199
200     STACK TYPE DG[STACK MARK IG] := LIT HELD M;
201
202 'COMMENT' PUT THE LITERAL SIZE ON THE STACK;
203
204     STACK DG[STACK MARK IG] := 4;(IMPLEMENTATION DEPENDENT)
205     EXIT M;
206
207 CASE 15 LL:
208     PTRACE M(TA PROCM,"ISTCHLIT");
209
210 'COMMENT' OPERATION CODE IS ZERO - OPERATION MODIFIER IS
211 "STACK CHARACTER LITERAL";
212
213     UPSTACK M;
214     BYTE STACK MARK IL := STACK MARK IG*4;
215
216     'FOR' KL :=RUN TIME PROG MARK IG+2 'STEP' 1 'UNTIL' RUN TIME PROG MARK IG+ PROGRAMDG[RUN
217     TIME PROG MARK IG+1]+1
218     'DO'
219     'BEGIN'
220        BYTE STACK DL[BYTE STACK MARK IL] := PROGRAM DG[KL];
221        INC M(BYTE STACK MARK IL,1);
222     'END';
223     STACK MARK IG := BYTE STACK MARK IL/4+(
224 'IF' 'INTEGER' (BYTE STACK MARK IL/4)*4=BYTE STACK MARK IL
225     'THEN' 0
226     'ELSE' 1)-1;
227
228 'COMMENT' PUT CHAR STRING LITERAL SIZE ON THE STACK AND MARK THE STACK
229 AS LITERAL HELD;
230
231     UPSTACK M;
232     STACK DG[STACK MARK IG] :=PROGRAM DG[RUN TIME PROG MARK IG+1];(STRING LENGTH)
233     STACK TYPE DG[STACK MARK IG] := LIT HELD M;
234     INC M(RUN TIME PROG MARK IG,PROGRAM DG[RUN TIME PROG MARK IG+1]+1);
235     EXIT M;
236
237 CASE 16 LL:
238     PTRACE M(TA PROC M,"IDCL");
239
240 'COMMENT' OPERATION CODE IS ZERO - OPERATION MODIFIER IS "DECLARE";
241
242
243 'COMMENT' FETCH THE SIZE OF THE OBJECT BEING DECLARED;
244
245     SIZE IL := OBJ SIZE M[OPERAND IL] ;
246
247 'COMMENT' CHECK MEMORY FOR OVERFLOW;
248
249
```

```
253         ERROR HANDLER PG('LITERAL'(R),"MEMORY%IS%FULL");
254      'END';
255
256   'COMMENT' PUT MEMORY ADDRESS INTO DATA TABLE;
257
258      OBJ TAB DG[OPERAND IL+1] := MEM MARK IG;
259      INC M(MEM MARK IG,SIZE IL);
260
261   'COMMENT' REMEMBER HOW MUCH MEMORY THE FUNCTION USED;
262
263      FUNC MEMORY DG[FUNC STC MARK IG-1] := FUNC MEMORY DG[FUNC STC MARK IG-1]+SIZE IL;
264      EXIT M;
265
266   CASE 2 LL:
267
268   'COMMENT' OPERATION CODE IS ONE;
269
270
271      'IF' OP MOD IL>4
272      'THEN'
273      'BEGIN'
274         ERROR HANDLER PG('LITERAL'(R),"OPERATION%MODIFIER%(1)%UNKNOWN");
275      'END'
276      'ELSE' 'GOTO' CASE 2 X SL[OP MOD IL+1];
277
278   CASE 2 FN CALL LL:
279      PTRACE M(TA PROC M,"IFNCALL");
280      FUNC STACK DG[FUNC STC MARK IG] := RUN TIME PROG MARK IG;(REMEMBER RETURN ADDRESS)
281      FUNC NAME DG[FUNC STC MARK IG] := STACK DG[STACK MARK IG];(REMEMBER WHICH FUNCTION IS CALLED)
282      FUNC MEMORY DG[FUNC STC MARK IG] := 0;(ZERO MEMORY USE INDICATOR)
283      INC M(FUNC STC MARK IG,1);
284
285   'COMMENT' CHECK FOR STACK OVERFLOW;
286
287
288      'IF' FUNC STC MARK IG>FUNC STC SIZE M
289      'THEN'
290      'BEGIN'
291         ERROR HANDLER PG('LITERAL'(R),"TOO%MANY%FUNCTIONS%HAVE%BEEN%CALLED");
292      'END';
293
294   'COMMENT' FETCH THE STACK VALUE AND CONVERT IT INTO "ADDRESS" FORMAT;
295
296      OBJ ADD IL := POP ADDRESS PG;
297
298   'COMMENT' CHECK FOR BUILT-IN ROUTINE;
299
300
301      'IF' OBJ ADD IL<0
302      'THEN' BUILT IN PG(-OBJ ADD IL)
303      'ELSE'
304      'BEGIN'
305
306   'COMMENT' ALTER PROGRAM COUNTER (AS THIS IS A MULE FUNCTION CALL) TO
307   TRANSFER CONTROL;
308
309         RUN TIME PROG MARK IG := OBJ ADD IL-1;
310      'END';
311      EXIT M;
312
313   CASE 2 DIRECT BRANCH LL:
314      PTRACE M(TA PROC M,"IGOTO");
315
```

```
319     EXIT M;
320
321 CASE 2 FN RETURN LL:
322     PTRACE M(TA PROC M,"IFNRET");
323
324 'COMMENT' A FUNCTION RETURN FROM A MULE FUNCTION - DOWN THE FUNCTION STACK;
325
326     FUNC STC MARK IG := FUNC STC MARK IG-1;
327
328 'COMMENT' AND THEN ALTER THE PROGRAM MEMORY COUNTER TO TRANSFER CONTROL;
329
330     RUN TIME PROG MARK IG := FUNC STACK DG[FUNC STC MARK IG] ;
331     MEM MARK IG := MEM MARK IG-FUNC MEMORY DG[FUNC STC MARK IG];
332 (RELINQUISH DYNAMICALLY ALLOCATED MEMORY)
333     EXIT M;
334
335 CASE 2 DESTACK LL:
336     PTRACE M(TA PROC M,"IDESTC");
337
338 'COMMENT' A DESTACKING OPERATION;
339
340
341 'COMMENT' DESTINATION OF THE DESTACK MUST BE AN ADDRESS - CHECK IT;
342
343
344     'IF' STACK TYPE DG[STACK MARK IG]<>ADDR HELD M
345     'THEN'
346     'BEGIN'
347        OBJ ADD IL := POP ADDRESS PG;
348        RH SIZE IL := 'BITS'[6,18]OBJ ADD IL;
349        OBJ ADD IL := 'BITS'[18,0]OBJ ADD IL;
350
351        'IF' RH SIZE IL=0
352        'THEN' ERRORHANDLER PG('LITERAL'(R),"DESTINATION%OF%AN%ASSIGNMENT%IS%NOT%AN%ADDRESS");
353     'END'
354     'ELSE'
355     'BEGIN'
356        RH SIZE IL := OBJ SIZE M[STACK DG[STACK MARK IG]]; (THE OBJECT SIZE OF
357    THE DESTINATION OF THE DESTACK)
358        OBJ ADD IL := POP ADDRESS PG ; (THE DESTINATION ADDRESS)
359     'END';
360     OPERAND IL := STACK TYPE DG[STACK MARK IG]; (FETCH THE TYPE OF THE SOURCE
361    OPERAND)
362     LH SIZE IL :=
363
364     'IF' OPERAND IL = LIT HELD M
365     'THEN' STACK DG[STACK MARK IG]
366     'ELSE'
367
368     'IF' OPERAND IL=VAL HELD M
369     'THEN' OBJ SIZE M[STACK DG[STACK MARK IG]]
370     'ELSE' DEFAULT OBJ SIZE M;(FIND OUT THE SIZE OF THE OBJECT - A LITERAL'S SIZE IS ON THE STACK)
371
372 'COMMENT' THE LEFT AND RIGHT HAND SIZES MUST BE THE SAME - CHECK IT
373  ..... BUT NOT IF THE OPERAND IS A CONSTANT AS A CONSTANT CAN
374  BE VARIABLE SIZE;
375
376
377     'IF' OPERAND IL <> LIT HELD M
378     'THEN'
379     'BEGIN'
380
381        'IF' LH SIZE IL<>RH SIZE IL
```

```
385          ;
386       'END';
387    'END';
388
389    'IF' OPERAND IL=LIT HELD M
390    'THEN'
391    'BEGIN'
392
393 'COMMENT' SOURCE IS A LITERAL SO COPY VALUE FROM THE STACK INTO MEMORY;
394
395       BYTE STACK MARK IL := (STACK MARK IG-LH SIZE IL/4-(
396 'IF' 'INTEGER' (LH SIZE IL/4)*4=LH SIZE IL
397       'THEN' 0
398       'ELSE' 1))*4;
399
400 'COMMENT' CHECK THAT THE CONSTANT VALUE IS NOT "POSSIBLY"
401 A SMALLER SIZE THAN THE OPERAND OTHERWISE RUBBISH WILL BE COPIED
402 FROM THE STACK INTO THE OPERANDS LOCATION;
403
404
405       'IF' RH SIZE IL > LH SIZE IL
406       'THEN'
407       'BEGIN'
408          OBJ ADD IL := OBJ ADD IL+(RH SIZE IL-LH SIZE IL);
409          RH SIZE IL := LH SIZE IL ;
410       'END'
411       'ELSE' BYTE STACK MARK IL := BYTE STACK MARK IL +(LH SIZE IL-RH SIZE IL);
412
413       'FOR' KL := OBJ ADD IL 'STEP' 1 'UNTIL' OBJ ADD IL+RH SIZE IL-1
414       'DO'
415       'BEGIN'
416          MEMORY DG[KL] := BYTE STACK DL[BYTE STACK MARK IL];
417          INC M(BYTE STACK MARK IL,1);
418       'END';
419       POP INTEGER PG; (REMOVE ITEM FROM THE STACK)
420    'END'
421    'ELSE'
422
423    'IF' OPERAND IL=VAL HELD M
424    'THEN'
425    'BEGIN'
426
427 'COMMENT' SOURCE OPERAND IS A VALUE HELD IN MEMORY - DO A MEMORY TO MEMORY COPY;
428
429
430       'FOR' KL :=OBJ TAB DG[STACK DG[STACK MARK IG]+1] 'STEP' 1 'UNTIL' OBJ TAB DG[STACKDG[
431       STACKMARKIG]+1]+LH SIZE IL-1'DO'
432       'BEGIN'
433          MEMORY DG[OBJ ADD IL] := MEMORY DG[KL];
434          INC M(OBJ ADD IL,1);
435       'END';
436       POP INTEGER PG; (REMOVE ITEM FROM THE STACK)
437    'END'
438    'ELSE'
439    'BEGIN'
440
441 'COMMENT' OPERAND IS AN ADDRESS SO STORE THE ADDRESS IN MEMORY;
442
443       LH SIZE IL := OBJ SIZE ML[STACK DG[STACK MARK IG]];(SIZE OF OBJECT REFERENCED)
444       OPERAND IL := POP ADDRESS PG;(ADDRESS OF OBJECT REFERENCED)
445
446 'COMMENT' AN ADDRESS REFERENCE IS MADE UP OF
447    BOTTOM 18BITS = ADDRESS
```

```
451         OPERAND IL := LH SIZE IL*262144+OPERAND IL;
452
453         'FOR' KL := DEFAULT OBJ SIZE M ,KL-1 'WHILE' KL>0
454         'DO'
455         'BEGIN'
456            MEMORY OG[OBJ ADD IL+KL-1] := BYTE 0 M(OPERAND IL);
457            OPERAND IL := OPERAND IL/64;
458         'END';
459      'END';
460      EXIT H;
461
462   ERROR LL:
463      ERROR HANDLER PG('LITERAL'(R),"UNTRAPPED%RUN %TIME%ERROR");(TEMPORARY MEASURE)
464
465   EXIT LL:
466      INC M(RUN TIME PROG MARK IG,1);
467
468      'IF' RUN TIME PROG MARK IG > PROG SIZE M
469      'THEN'
470      'BEGIN'
471         ERROR HANDLER PG('LITERAL'(R),"PROGRAM %MEMORY%LIMIT%VIOLATION");
472      'END';
473      REPEAT M(START LL);
474
475   RETURN LL:
476   'END'INTERPRET PG;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.MAPPG(1/) PRODUCED ON  4MAR82 AT 17.11.02

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.57.26 USING I381

DOCUMENT    MAPPG(1/)

'PROCEDURE' MAP LIST PG('VALUE''INTEGER' Q1,Q2);
'BEGIN'
  OINT M(Q1);
  OSP M(1);
  OTEXT M(Q2);
  ONL M;
'END' MAP LIST PG;
'PROCEDURE' MAP PG;
'BEGIN'
OSEL M(OMAPF M);
MAP LIST PG('LOCATION'(ELEMENT IG),"ELEMENT IG");
MAP LIST PG('LOCATION'(ELEMENT TYPE IG),"ELEMENT TYPE IG");
MAP LIST PG('LOCATION'(IRRELAVANT SKIPPED BG),"IRRELAVANT SKIPPED BG");
MAP LIST PG('LOCATION'(CURRENT LINE IG),"CURRENT LINE IG");
MAP LIST PG('LOCATION'(OBJ TAB MARK IG),"OBJ TAB MARK IG");
MAP LIST PG('LOCATION'(MAX OBJ TABLE MARK IG),"MAX OBJ TABLE MARK IG");
MAP LIST PG('LOCATION'(TA PROC PRINT IG),"TA PROC PRINT IG");
MAP LIST PG('LOCATION'(LATEST PROCEDURE IG),"LATEST PROCEDURE IG");
MAP LIST PG('LOCATION'(PROG MARK IG),"PROG MARK IG");
MAP LIST PG('LOCATION'(KG),"KG");
MAP LIST PG('LOCATION'(FUNC NEST DEPTH IG),"FUNC NEST DEPTH IG");
MAP LIST PG('LOCATION'(RUN TIME PROG MARK IG),"RUN TIME PROG MARK IG");
MAP LIST PG('LOCATION'(START HERE IG),"START HERE IG");
MAP LIST PG('LOCATION'(STACK MARK IG),"STACK MARK IG");
MAP LIST PG('LOCATION'(FUNC STC MARK IG),"FUNC STC MARK IG");
MAP LIST PG('LOCATION'(MEM MARK IG),"MEM MARK IG");
MAP LIST PG('LOCATION'(CHAR BUFF MARK IG),"CHAR BUFF MARK IG");
MAP LIST PG('LOCATION'(IN INTRO BG),"IN INTRO BG");
MAP LIST PG('LOCATION'(SIZE OF OBJECT IG),"SIZE OF OBJECT IG");
MAP LIST PG('LOCATION'(DUMMY),"DUMMY");
MAP LIST PG('LOCATION'(OBJECT STRING DG[0]),"OBJECT STRING DG[0:OBJ STR SIZE M]");
MAP LIST PG('LOCATION'(PROGRAM DG[0]),"PROGRAM DG[0:PROG SIZE M]");
MAP LIST PG('LOCATION'(STACK TYPE DG[0]),"STACK TYPE DG[0:STACK SIZE M]");
MAP LIST PG('LOCATION'(MEMORY DG[0]),"MEMORY DG[0:MEMSIZE M]");
MAP LIST PG('LOCATION'(CHARACTER BUFFER DG[0]),"CHARACTER BUFFER DG[0:120]");
MAP LIST PG('LOCATION'(DUMMYA[0]),"DUMMYA[0:1]");
MAP LIST PG('LOCATION'(OBJ TAB DG[0]),"OBJ TAB DG[0:OBJ TABLE SIZE M]");
MAP LIST PG('LOCATION'(STACK DG[0]),"STACK DG[0:STACK SIZE M]");
MAP LIST PG('LOCATION'(FUNC MEMORY DG[0]),"FUNC MEMORY DG[0:FUNC STC SIZE M]");
MAP LIST PG('LOCATION'(FUNC STACK DG[0]),"FUNC STACK DG[0:FUNC STC SIZE M]");
MAP LIST PG('LOCATION'(FUNC NAME DG[0]),"FUNC NAME DG[0:FUNC STC SIZE M]");
MAP LIST PG('LOCATION'(FUNCTION START DG[0]),"FUNCTION START DG[0:FUNC STC SIZE M]");
MAP LIST PG('LOCATION'(DUMMYIA[0]),"DUMMYIA[0:1]");
MAP LIST PG('LOCATION'(POWER OF 10 DG[0]),"POWER OF 10 DG[0:6] ");
MAP LIST PG('LOCATION'(POWER OF 2 DG[0]),"POWER OF 2 DG[0:15]");
MAP LIST PG('LOCATION'(DUMMYPIA[0]),"DUMMYPIA[0:1]");
'END' MAP PG;

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.LEXISPG(37/)  PRODUCED ON 13AUG82 AT 08.44.09

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.00.18 USING I381

DOCUMENT    LEXISPG(37/)


'RECURSIVE' LEXIS PG;

'COMMENT'
PURPOSE
  THIS PROCEDURE IS THE START UP PROCEDURE OF THE MULE ENVIRONMENT
  IT CALLS THE PROCEDURE 'FETCH ELEMENT PG' WHICH RETURNS AN
  OBJECT TABLE INDEX NUMBER REPRESENTING THE ELEMENT READ. SOME
  ELEMENTS ARE BUILT-IN TO THE MULE ENVIRONMENT AND HENCE RETURN A
  KNOWN INDEX VALUE.  THIS PROCEDURE RELIES UPON THE FOLLOWING :
    OBJECT TABLE MARK             ELEMENT
    -----------------             -------
            4                     USE
            6                     AN OPEN ROUND BRACKET
            8                     FUNCTION
            10                    ->
            12                    .
            14                    @
            16                  END
         18                   A CLOSED ROUND BRACKET

  THIS PROCEDURE STEERS CONTROL TO THE FOLLOWING SECON LEVEL PROCEDURES
  ON RECEIVING THE ELEMENTS ABOVE :
     ELEMENT                     PROCEDURE CALLED
     -------                     ----------------
     USE                         INTRODUCTION PG
     AN OPEN ROUND BRACKET       INFIX PG
     FUNCTION                    FUNCTION DEF PG
     ->                          DESTACK PG
     ()                           FUNCTION CALL PG
     @                           STACK PG
     CONSTANT VALUE              STACK PG
     OBJECT NAME                 STACK PG
     END                         END FUNCTION DEF PG
     OTHER                       ERROR HANDLER PG, BUT IGNORES COLON DOT AND RIGHT BRACKET
GLOBAL VARIABLES
  ELEMENT IG (R)
  ELEMENT TYPE IG (R)
  FUNC NEST DEPTH IG
LOCAL VARIABLES
  SELECT SL (SELECTION SWITCH)
 REMAIN IN LEXIS KL (A COUNTER WHICH IF NON-ZERO CAUSES THE PROCEDURE
                         TO LOOP UNTIL A MATCHING FUNCTION END IS RECEIVED)
MACROS
  ELE LITERAL M
  ELE CHAR LIT M
  ELE OPERAND M
  TA PROC M
  EXIT M
  RAT ST ADD M
  RAT ST VAL M
  RAT ST LIT M
```

```
   FUNCTION DEF PG
   DESTACK PG
   FUNCTION CALL PG
   STACK ADDRESS PG
   STACK LITERAL PG
   STACK VALUE PG
   ERROR HANDLER PG
   FETCH ELEMENT PG
MESSAGES PRODUCED
   1  SYNTAX ERROR, UNKNOWN ELEMENT ..
METHOD
   THE PROCEDURE SWITCHES ON THE ELEMENT RECEIVED USING THE ELEMENT
   TYPE TO TELL IF THE ELEMENT READ WAS A LITERAL OR NOT.
;

'BEGIN'
   'INTEGER' REMAIN IN LEXIS KL;
   'SWITCH' SELECT SL:=COLON LL,
   EXIT LL,
   USE LL,
   OPEN BRACKET LL,
   FUNCTION LL,
   RIGHT ARROW LL,
   LEFT RIGHT BRACKET,
   AT LL,
   END LL,
   EXIT LL;
   PTRACE M(TA PROC M,"LEXIS");
   REMAIN IN LEXIS BL := FALSE M;

START LL:

   'IF' FUNC NEST DEPTH IG = -1 'AND' PROGRAM DG[START HERE IG] <>0
   'THEN' INTERPRET PG;(IF WE ARE NOT IN A FUNCTION DEFINITION AND THERES SOMETHING IN
 PROGRAM MEMORY WE MUST WANT TO INTERPRET IT)
   FETCH ELEMENT PG ;(CAUSES AN ELEMENT TO BE FETCHED & GLOBAL VARIABLES 'ELEMENT IG' &
   'ELEMENT TYPE IG' TO BE WRITTEN TO AS RETURNED VALUES )

   'IF' ELEMENT IG <= 18 'AND' ELEMENT TYPE IG=ELE.OPERAND M
   'THEN' 'GOTO' SELECT SL[ELEMENT IG/2+1]
   'ELSE'
   'BEGIN'

'COMMENT'
            THIS IS THE CASE WHERE THE ELEMENT IS EITHER A LITERAL OR AN
             OPERAND. IF
            IT IS NEITHER OF THESE THEN THE PROGRAM IS IN ERROR & AN ERR
            OR MESSAGE
            WILL BE GENERATED;


      'IF' ELEMENT TYPE IG = ELE LITERAL M
      'THEN'
      'BEGIN'
         PTRACE M(TA PROC M,"      STACK LITERAL PG");
         STACK PG(RAT ST LIT M);
      'END'
      'ELSE'

      'IF' ELEMENT TYPE IG=ELE CHAR LIT M
      'THEN'
      'BEGIN'
         PTRACE M(TA PROC M,"STACCHLIT");
```

```
      'IF' ELEMENT TYPE IG = ELE OPERAND M
      'THEN'
      'BEGIN'
         PTRACE M(TA PROC M," STACK VALUE PG");
         STACK PG(RAT ST VAL M);
      'END'
      'ELSE' ERROR HANDLER PG(1,"SYNTAX%ERROR%-%UNKNOWN%ELEMENT");
    'END';
    EXIT M;

'COMMENT' SIMULATED CASE STATEMENT DEALING WITH ALL OTHER POSSIBLE
         ELEMENTS;


USE LL:
   PTRACE M(TA PROC M,"INTRODUCTION PG");
   INTRODUCTION PG;
   EXIT M;

OPEN BRACKET LL:
   PTRACE M(TA PROC M,"INFIX PG");
   INFIX PG;
   EXIT M;

FUNCTION LL:
   PTRACE M(TA PROC M,"FUNCTION DEF PG");
   INC M(REMAIN IN LEXIS KL,1); (CAUSES CONTROL TO REMAIN IN LEXISPG)
   FUNCTION DEF PG;
   EXIT M;

RIGHT ARROW LL:
   PTRACE M(TA PROC M,"DESTACK PG");
   DESTACK PG;
   EXIT M;

LEFT RIGHT BRACKET LL:
   PTRACE M(TA PROC M,"FUNCTION CALL PG");
   FUNCTION CALL PG;
   EXIT M;

AT LL:
   PTRACE M(TA PROC M,"STACK ADDRESS PG");
   FETCH ELEMENT PG;
   STACK PG(RAT ST ADD M);
   EXIT M;

COLON LL:
   PTRACE M(TA PROC M,"COLON");
   EXIT M;

END LL:
   PTRACE M(TA PROC M,"ENDFUNC");
   REMAIN IN LEXIS KL := REMAIN IN LEXIS KL -1; (CAUSES RELINQUISH OF CONTROL FROM LEXISPG WHEN ZERO)
   END FUNCTION DEF PG;
   EXIT M;

EXIT LL:

   'IF' REMAIN IN LEXIS BL <> FALSE M
   'THEN' REPEAT M(START LL);
'END'LEXIS PG;
```

Ā***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT

TING OF :ZZZZZZZZZZZN.END(22/)  PRODUCED ON 31MAR82 AT 17.32.44

PUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.53.23 USING I381

MENT    END(22/)


'ENTRY'(28);
NEWLINE;
'CODE'
'BEGIN'
   SUSWT('LITERAL'(CL))
'END';
'ENTRY'(20);

'FOR' KG := 0 'STEP' 1 'UNTIL' OBJ STR SIZE M
'DO' OBJECT STRING DG[KG] := EMPTY M;
CURRENT LINE IG := 0;
FUNC NEST DEPTH IG := -1;
OSEL M(0);
START 0 LL:
LEXIS PG;
REPEAT M(START 0 LL);
MEM MARK IG := 0;
'CODE'
'BEGIN'
   SUSWT('LITERAL'(LE))
'END';
'ENTRY'(21);

'COMMENT' TO REFRESH VARIABLES AFTER INITIALISATION OF THE LANGUAGE SYMBOLS ;
OBJ TAB DG[21] := -1;
PROGMARK IG := 1;
STACK MARK IG := -1;
START HERE IG := 1;
FUNC NEST DEPTH IG := -1;
MEM MARK IG := 0;

'ENTRY'(22);
RESTART LC:
CURRENT LINE IG := 0 ;
RUN TIME PROG MARK IG := START HERE IG;
PROGRAM DG[RUN TIME PROG MARK IG] := 0;
ISEL M(0);
FUNC STC MARK IG := 0;
START 1 LL:
LEXIS PG;
REPEAT M(START 1 LL);
'CODE'
'BEGIN'
   SUSWT('LITERAL'(OK))
'END';
'ENTRY'(27);
MAP PG;
ONL M;
'CODE'
'BEGIN'
   SUSWT('LITERAL'(MA))
'END';
'END'
'FINISH'
'STOP'


*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT6*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3
```

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.LANGSYMB(10/INIT) PRODUCED ON 17FEB82 AT 19.37.30

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.00.53 USING I381

DOCUMENT    LANGSYMB(10/INIT)


      0 :
      1 .
      2 USE
      3 (
      4 FUNCTION
      5 =>
      6 ()
      7 @
      8 END
      9 )
     10 [BUILT IN ROUTINES FOLLOW]
     11 MAKEBUILTIN
     12 INCHAR
     13 OUTCHAR
     14 +
     15 -
     16 *
     17 /
     18 ^
     19 <
     20 >
     21 <=
     22 >=
     23 =
     24 <>
     25 AND
     26 OR
     27 UNALLOCATED
     28 IF
     29 GOTO
     30 POP
     31 STACKNAME
     32 OUTINT
     33 COMPILE
     34 SUBSTR
     35 VALUE
     36 .
     37 :

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.BUILTINS(8/MULE) PRODUCED ON 17FEB82 AT 19.38.09

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.58.35 USING I381

DOCUMENT BUILTINS(8/MULE)

```
INCHAR,2 MAKEBUILTIN()
OUTCHAR,3 MAKEBUILTIN()
+,4 MAKEBUILTIN()
-,5 MAKEBUILTIN()
*,6 MAKEBUILTIN()
/,7 MAKEBUILTIN()
^,8 MAKEBUILTIN()
<,9 MAKEBUILTIN()
>,10 MAKEBUILTIN()
<=,11 MAKEBUILTIN()
>=,12 MAKEBUILTIN()
=,13 MAKEBUILTIN()
<>,14 MAKEBUILTIN()
AND,15 MAKEBUILTIN()
OR,16 MAKEBUILTIN()
UNALLOCATED,17 MAKEBUILTIN()
IF,18 MAKEBUILTIN()
GOTO,19 MAKEBUILTIN()
POP,20 MAKEBUILTIN()
STACKNAME,21 MAKEBUILTIN()
OUTINT,22 MAKEBUILTIN()
COMPILE,23 MAKEBUILTIN()
SUBSTR,24 MAKEBUILTIN()
VALUE,25 MAKEBUILTIN()
:
:
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^
£LISTING OF :ZZZZZZZZZZZH.TESTA(6/MULE) PRODUCED ON 28MAY81 AT 19.25.04
£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.57.39 USING I381
DOCUMENT    TESTA(6/MULE)

[
PURPOSE
  TO TEST THE OBJECT INTRODUCTION FACILITY TO CHECK THAT IT OPERATES
  AS DEFINED.
RELIES UPON
  NO OTHER TEST
TEST CODE
]
USE AAAA .
USE AAAAA,AAAAB .
USE AAABA,AAAABB,CAAAA .
USE BBBB:5 .
USE BBBBB : 7,BCDE777:12 .
USE ALPHABETICAND1234567890:2 .
USE ££££ .
USE £I$() : 3,XYZ:4,// : 1 .
[
EXPECTED OUTPUT
STARTING AT OBJTABDG(62) FOR A RANGE OF 28 ELEMENTS
    11573   0P3+    +196827   £00600333   STOC  0  219
    11574   001-        +93   £00000135   LDX   0  93
    11575   0P3C    +196835   £00600343   STOC  0  227
    11576   001@        +96   £00000140   LDX   0  96
    11577   0P3L    +196844   £00600354   STOC  0  236
    11578   001C        +99   £00000143   LDX   0  99
    11579   0P3U    +196853   £00600365   STOC  0  245
    11580   001F       +102   £00000146   LDX   0  102
    11581   0P3^    +196862   £00600376   STOC  0  254
    11582   001I       +105   £00000151   LDX   0  105
    11583   0P48    +196872   £00600410   STOC  0  264
    11584   001L       +108   £00000154   LDX   0  108
    11585   1 4!    +327953   £01200421   LDCH  0  273
    11586   0010       +111   £00000157   LDX   0  111
    11587   1P4)    +459033   £01600431   DCH   0  281
    11588   001T       +116   £00000164   LDX   0  116
    11589   304B    +786722   £03000442   BUX   0  290
    11590   001[       +123   £00000173   LDX   0  123
    11591   0@4M    +131373   £00400455   STO   0  301
    11592   0027       +135   £00000207   LDX   0  135
    11593   0P58    +196936   £00600510   STOC  0  328
    11594   0029       +137   £00000211   LDX   0  137
    11595   0P5     +196944   £00600520   STOC  0  336
    11596   002<       +140   £00000214   LDX   0  140
    11597   1J5)    +262489   £01000531   ANDX  0  345
    11598   002?       +143   £00000217   LDX   0  143
    11599   U 5@     +65888   £00200540   LDXC  0  352
    11600   002£       +147   £00000223   LDX   0  147
]
:

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^
```

Â*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZH.TESTB(11/MULE)   PRODUCED ON 20JAN82 AT 18.33.20

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.58.16 USING I381

DOCUMENT   TESTB(11/MULE)

```
[*******************************************************************
TESTA
**************************************************************************]
[
PURPOSE
  TO TEST THE OBJECT INTRODUCTION FACILITY TO CHECK THAT IT OPERATES
  AS DEFINED.
RELIES UPON
  NO OTHER TEST
TEST CODE
]
USE AAAA .
USE AAAAA,AAAAB .
USE AAABA,AAAABB,CAAAA .
USE BBBB:5 .
USE BBBBB : 7,BCDE777:12 .
USE ALPHABETICAND1234567890:2 .
USE ££££ .
USE £!$() : 3,XYZ:4,// : 1 .
[
EXPECTED OUTPUT
STARTING AT OBJTABDG(62) FOR A RANGE OF 28 ELEMENTS
    11573    0P3+      +196827   £00600333    STOC  0  219
    11574    001-          +93   £00000135    LDX   0  93
    11575    0P3C      +196835   £00600343    STOC  0  227
    11576    001@          +96   £00000140    LDX   0  96
    11577    0P3L      +196844   £00600354    STOC  0  236
    11578    001C          +99   £00000143    LDX   0  99
    11579    0P3U      +196853   £00600365    STOC  0  245
    11580    001F         +102   £00000146    LDX   0  102
    11581    0P3Â      +196862   £00600376    STOC  0  254
    11582    001I         +105   £00000151    LDX   0  105
    11583    0P48      +196872   £00600410    STOC  0  264
    11584    001L         +108   £00000154    LDX   0  108
    11585    1 41      +327953   £01200421    LDCH  0  273
    11586    0010         +111   £00000157    LDX   0  111
    11587    1P4)      +459033   £01600431    DCH   0  281
    11588    001T         +116   £00000164    LDX   0  116
    11589    304B      +786722   £03000442    BUX   0  290
    11590    001[         +123   £00000173    LDX   0  123
    11591    0@4M      +131373   £00400455    STO   0  301
    11592    0027         +135   £00000207    LDX   0  135
    11593    0P58      +196936   £00600510    STOC  0  328
    11594    0029         +137   £00000211    LDX   0  137
    11595    0P5       +196944   £00600520    STOC  0  336
    11596    002<         +140   £00000214    LDX   0  140
    11597    105)      +262489   £01000531    ANDX  0  345
    11598    002?         +143   £00000217    LDX   0  143
    11599    0 5@       +65888   £00200540    LDXC  0  352
    11600    002£         +147   £00000223    LDX   0  147
]
:
```

```
55 [
56 PURPOSE
57   TO TEST THE INFIX STACKING SYNTAX NOTATION
58 RELIES UPON
59   TESTA RUNNING SUCCESFULLY
60 TEST CODE
61 ]
62
63
64
65
66
67
68
69 [ NOW INITIALISE THE VARIABLES SO THAT WE CHAN CHECK THE RESULT]
70 1 -> @AAAA
71 2 -> @BBBB
72 FUNCTION 3 END -> @AAAAA
73 FUNCTION 4 END -> @CAAAA
74 5 -> @ £1$()
75 6 -> @ALPHABETICAND1234567890
76 7 -> @BCDE777
77 8 -> @AAAABB
78 (AAAA+BBBB); OUTINT();
79 ( @AAAA+ @BCDE777); OUTINT();
80 (12 +1270); OUTINT();
81 (AAAAA() + CAAAA() ); OUTINT();
82 (FUNCTION(AAAA+BBBB)END() +FUNCTION( £1$() + ALPHABETICAND1234567890)END() );
83 OUTINT();
84 ( (AAAA*BBBB) - (BCDE777*AAAABB) ); OUTINT();
85 [
86 EXPECTED OUTPUT
87 3
88 ?
89 1282
90 7
91 14
92 -54
93 ]
94 :
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZZH.IFELSE(1/MULE)   PRODUCED ON 24MAR82 AT 18.01.12

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.59.27 USING I381

DOCUMENT   IFELSE(1/MULE)

```
FUNCTION
[
IFELSE
  PURPOSE
    TO PROVIDE THE "IF...THEN...ELSE" OF CONVENTIONAL PROGRAMMING LANGUAGES.
  PARAMETERS
    1. THE RESULT OF A "BOOLEAN" EXPRESSION
    2. A FUNCTION TO BE APPLIED IF THE FIRST PARAMETER
       EVALUATES TO NON ZERO.
    3. A FUNCTION TO BE APPLIED OTHERWISE.
  STACK
    THE STACK IS LEFT AS IT WAS EXCEPT FOR THE REMOVAL OF THE 3 PARAMETERS
    SPECIFIED ABOVE.
]
  USE TRUE,FALSE.
  -> @FALSE ; -> @TRUE ;

    FUNCTION
      TRUE(); POP();
    END ;  IF() ;
    [THE CONDITION IS ON THE STACK ALREADY & IF IT EVALUATES TO NON ZERO.....
     THE FUNCTION "TRUE()" WILL BE CALLED FOLLOWED BY "POP()". THIS
     WILL CAUSE BOTH THE IF-FUNCTION AND THE IF-ELSE-FUNCTIONS TO BE ..
     RETURNED]

  FALSE() ; [IF WE GET HERE THE ANSWER MUST HAVE BEEN FALSE ]

END -> @IFELSE ;
:
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZZH.WHILE(4/MULE) PRODUCED ON 25MAR82 AT 08.12.13

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 10.59.04 USING I381

DOCUMENT   WHILE(4/MULE)

```
FUNCTION
[
WHILE
  PURPOSE
    TO PROVIDE THE WHILE FACILITY WHEREBY A LOOP IS EXECUTED FOR AS LONG AS
    A CONDITION EVALUATES TO NON ZERO.
  PARAMETERS
    1. A FUNCTION WHICH REPRESENTS THE WHILE CONDITION. THE FUNCTION WILL BE
       EXECUTED FOR EACH LOOP EXECUTION AND IS EXPECTED TO DELIVER A VALUE
       TO THE LOOP.
    2. A FUNCTION WHICH REPRESENTS THE LOOP BODY.
  STACK
    THE STACK IS ,LEFT UNCHANGED EXCEPT FOR THE REMOVAL OF THE 2 PARAMETERS.
]
  USE BODY,CONDITION,LOOP.
  -> @BODY  -> @CONDITION ;
    FUNCTION
      CONDITION() ;
      BODY; FUNCTION POP(); POP()  END ;  IFELSE();
      POP();
      LOOP() ;
    END -> @LOOP;
  LOOP();
END-> @WHILE ;
:
```

^***BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*B**^

```
^*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3*BT3**^

£LISTING OF :ZZZZZZZZZZZN.ACCEPTANCE(31/MULE)   PRODUCED ON 16AUG82 AT 08.25.14

£OUTPUT BY LISTFILE IN ':TLFAK-10.RPSMITH' ON 16AUG82 AT 11.01.39 USING I381

DOCUMENT    ACCEPTANCE(31/MULE)

      0 [UTILITY FUNCTIONS USED WITHIN THE TESTS ]
      1 USE OUTTEXT,NL.
      2    FUNCTION
      3      USE LOCALBUFFER:62.
      4      "                                                    "-> @LOCALBUFFER;
      5      -> @LOCALBUFFER ; [READ PARAMETER FROM THE STACK]
      6      @LOCALBUFFER OUTCHAR() ;
      7    END -> @OUTTEXT;
      8
      9    FUNCTION
     10      USE LOCALNL:1.
     11      63-> @LOCALNL ;
     12      @LOCALNL OUTCHAR();
     13    END-> @NL;
     14
     15 "MULE ACCEPTANCE TESTS
     16 " OUTTEXT() ;
     17 [
     18 MULE ACCEPTANCE TEST TO TEST THE FOLLOWING
     19 ]
     20 [BUILT IN ROUTINES]
     21   [INPUT OUTPUT]
     22   "INPUT OUTPUT TESTS
     23 " OUTTEXT() ;
     24
     25   USE BUFFER:10 .
     26   @BUFFER INCHAR() :ABCDEFGHI
     27   [INCHAR READS 'ABCDEFGHI<EOL>' FROM THE INPUT STREAM &
     28    PUTS IT INTO THE BUFFER ]
     29   @BUFFER OUTCHAR(); NL(); [**E0 ABCDEFGHI]
     30   1275 OUTINT() ; [**E0 1275]
     31
     32   [DIADIC OPERATORS]
     33   "DIADIC TESTS
     34 " OUTTEXT() ;
     35   USE LH,RH .
     36   150-> @LH ; 50-> @RH ; [INITIALISE VARIABLES]
     37   (LH+RH) OUTINT() ; [**E0 200]
     38   (LH-RH) OUTINT() ; [**E0 100]
     39   (LH*RH) OUTINT() ; [**E0 7500]
     40   (LH/RH) OUTINT() ; [**E0 3]
     41   (LH<RH) OUTINT() ; [**E0 0]
     42   (LH>RH) OUTINT() ; [**E0 1]
     43   (LH<=RH) OUTINT() ; [**E0 0]
     44   (LH>=RH) OUTINT() ; [**E0 1]
     45   (LH=RH) OUTINT() ; [**E0 0]
     46   (LH<>RH) OUTINT() ; [**E0 1]
     47   (LH AND RH) OUTINT() ; [**E0 18]
     48   (LH OR RH) OUTINT() ; [**E0 182]
     49
     50   [CONDITIONALS]
     51   "CONDITIONAL TESTS
```

```
55      FUNCTION
56        "OK
57 "          -> @BUFFER ;
58        @BUFFER OUTCHAR();
59      END -> @OK;
60
61      FUNCTION
62        "FAIL
63 "          -> @BUFFER ;
64        @BUFFER OUTCHAR() ;
65      END -> @FAIL;
66
67    1,OK IF() ; [**EO OK]
68    0,FAIL IF() ; [NO OUTPUT]
69    LH,OK IF() ; [**EO OK]
70    RH,OK IF() ; [**EO OK]
71    @LH,OK IF() ; [**EO OK]
72    (LH>RH),OK IF() ; [**EO OK]
73    (LH>=RH),OK IF() ; [**EO OK]
74    (LH<RH),FAIL IF() ; [NO OUTPUT]
75    ( (LH AND RH) = (LH AND RH) ),OK IF() ; [**EO OK]
76
77    [GOTO IS NOT TESTED]
78
79    [POP]
80    "POP TESTS
81 " OUTTEXT() ;
82    USE INNERFUNCTION,OUTERFUNCTION .
83
84      FUNCTION
85        POP() ;
86      END -> @INNERFUNCTION ;
87
88      FUNCTION
89        INNERFUNCTION() ;
90        FAIL();
91      END -> @OUTERFUNCTION ;
92
93    OUTERFUNCTION() ;
94    OK() ; [**EO OK]
95
96    [STACK NAME]
97    "STACK NAME TESTS
98 " OUTTEXT();
99    USE NEWLINE:1,PRINTNAME .
100   63 -> @NEWLINE ;
101
102     FUNCTION
103     "          " -> @BUFFER;
104       STACKNAME() -> @BUFFER ;
105       @BUFFER OUTCHAR() ;
106       @NEWLINE OUTCHAR() ;
107     END -> @PRINTNAME ;
108
109   BUFFER PRINTNAME() ; [**EO BUFFER]
110   LH PRINTNAME() ; [**EO LH]
111   RH PRINTNAME() ; [**EO RH]
112   OK PRINTNAME() ; [**EO OK]
113   FAIL PRINTNAME() ; [**EO FAIL]
114   INNERFUNCTION PRINTNAME() ; [**EO INNERFUNCTION]
115   OUTERFUNCTION PRINTNAME() ; [**EO OUTERFUNCTION]
116   NEWLINE PRINTNAME() ; [**EO NEWLINE]
117   PRINTNAME PRINTNAME() ; [**EO PRINTNAME]
```

```
" OUTTEXT() ;
  "ABC123DEF4" -> @BUFFER ;
  USE STRING1:1,STRING3:3,STRING5:5 .

  @BUFFER,3,4 SUBSTR() ; OUTINT() ; [**EO 1048576]
  @BUFFER,3,1 SUBSTR() VALUE() OUTTEXT() ; NL();
[**EO 1]
  @BUFFER,6,3 SUBSTR() VALUE() OUTTEXT() ; NL();
[**EO DEF]
  @BUFFER,5,2 SUBSTR() VALUE() OUTTEXT() ; NL();
[**EO 3D]
  "XXX" @BUFFER,3,3 -> ( SUBSTR() ) ;
  @BUFFER OUTCHAR() ; NL(); [**EO ABCXXXDEF4]

  [COMPILE
    - THIS ADDS THE WHILE AND ELSE FACILITY TOO ]
  "COMPILE TESTS
" OUTTEXT() ;
USE IFELSE,WHILE.
  "IFELSE(/MULE)" COMPILE() ;
"WHILE(/MULE)" COMPILE() ;

  [IFELSE]
  "IF ELSE TESTS
" OUTTEXT();
  "          "-> @BUFFER;
  1,OK,FAIL IFELSE() ; [**EO OK]
  0,FAIL,OK IFELSE() ; [**EO OK]
  LH,OK,FAIL IFELSE() ; [**EO OK]
  RH,OK,FAIL IFELSE() ; [**EO OK]
  @LH,OK,FAIL IFELSE() ; [**EO OK]
  (LH>RH),OK,FAIL IFELSE() ; [**EO OK]
  (LH>=RH),OK,FAIL IFELSE() ; [**EO OK]
  (LH<RH),FAIL,OK IFELSE() ; [**EO OK]
  ( (LH AND RH) = (RH AND LH) ),OK,FAIL IFELSE() ; [**EO OK]

  [WHILE]
  "WHILE TESTS
 "OUTTEXT() ;
  USE LOOPCOUNTER,CHARACTER:1 .
  (0-1) -> @LOOPCOUNTER ; "ABCXXXDEF4"-> @BUFFER;
    FUNCTION

      (LOOPCOUNTER+1) -> @LOOPCOUNTER ;
      (LOOPCOUNTER < 10)
    END,
    FUNCTION

      @BUFFER,1,LOOPCOUNTER SUBSTR() VALUE() -> @CHARACTER ;
      @CHARACTER OUTCHAR() ;
    END WHILE() ;
  @NEWLINE OUTCHAR() ; [**EO ABCXXXDEF4]

  [INFIX STACKING OPERATION]
  "INFIX STACKING OPERATION TESTS
" OUTTEXT() ;
  USE $,!,£,$!,!!,£! .
  USE FNLH,FNRH .
  25 -> @LH ; 30 -> @RH ;
    FUNCTION 172 END -> @FNLH ;
    FUNCTION 3 END -> @FNRH ;
  1 -> @ $ ;
  2 -> @ ! ;
```

```
6 -> @ £1 ;
( $ + 1 ) ; OUTINT() ; [**E0 3]
(LH+RH) OUTINT() ; [**E0 55]
( ( $ + 1 ) + (LH+RH) ) OUTINT() ; [**E0 58]
( ( ( $ + 1 ) + (LH+RH) ) + £ ) OUTINT() ; [**E0 61]
( ( ( ( $ + 1 ) + (LH+RH) ) + £ ) + FNLH() ) OUTINT() ; [**E0 233]
( ( ( ( $ + 1 ) + (LH+RH) ) + £ ) + (FNLH() + FNRH() ) ) OUTINT() ;
[**E0 236]
( ( ( ( ( $ + 1 ) + (LH+RH) ) + £ ) + (FNLH() + FNRH() ) ) +
  FUNCTION 1, FUNCTION 3 END, FAIL IFELSE() END() ) OUTINT()  ; [**E0 239]
( ( ( 1+3 ) *2 ) -8) ; OUTINT() ;[**E0 0]
( (2* (1+3) ) -8) ; OUTINT(); [**E0 0]
( 8- (2* (1+3) ) ); OUTINT() ; [**E0 0]

[FUNCTIONS]
"FUNCTION TESTS
" OUTTEXT() ;
FUNCTION
  USE F1,F2,F3,F4.
    FUNCTION
      F2 PRINTNAME() ;
      F3() ;
    END -> @F2;
  FUNCTION
    F3 PRINTNAME() ;
    F4();
  END -> @F3;
  FUNCTION
    OK() ;
  END -> @F4;
F2();
END(); [**E0 F2]
       [**E0 F3]
       [**E0 OK]

USE RECURSE.
0-> @LOOPCOUNTER;
FUNCTION
  USE ENTER:1,EXIT:1.
  ENTER PRINTNAME();
  (LOOPCOUNTER+1) -> @LOOPCOUNTER;
  (LOOPCOUNTER<=4)
    FUNCTION RECURSE() END IF();
EXIT PRINTNAME() ;
END -> @RECURSE;
  RECURSE() ;
[**E0 ENTER]
[**E0 ENTER]
[**E0 ENTER]
[**E0 ENTER]
[**E0 ENTER]
[**E0 EXIT]
[**E0 EXIT]
[**E0 EXIT]
[**E0 EXIT]
[**E0 EXIT]

[END OF TEST]
"TEST ENDED
" OUTTEXT() ;
:
:
```